Execution Model and Optimizing Compilation for Execution Migration

by

Ilia Andreevich Lebedev

B.S., University of California, Berkeley (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author ....
Department of Electrical Engineering and Computer Science
August 21, 2013

Certified by..................
Srinivas Devadas
Professor
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Professor
Chair, Department Committee on Graduate Theses

Execution Model and Optimizing Compilation for Execution Migration

by

Ilia Andreevich Lebedev

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Although systems with hardware support for fine-grained execution migration are becoming a reality, no concrete execution model or compiler exist for these machines. This limits the complexity of software that can be written for these machines, and therefore also the scope of studies for which these machines can be used. In this thesis, we define a productive programming model for an execution migration platform by exposing migration as a set of interfaces usable with the C programming language via a custom optimizing compiler. We employ hardware-software co-design to describe a stack core architecture with support for partial context migration in order to simplify the compiler problem and improve compiler efficiency. We also consider instruction encoding in abstract terms to establish a baseline comparison of encoded instruction density to an ideal upper bound. The stack-based execution migration platform offers a new and unexplored cost model, which leads us to reevaluate the trade-offs associated with compilation for these architectures, and to explore novel algorithms, or novel applications of existing optimizations. Throughout this work, we attempt to gain a deep understanding of the costs and benefits of execution migration by aggressive design space exploration. We use the insight gained to better inform the the problem of compiling to this unorthodox architecture, and design the compiler, a library of optimized parallel primitives, and a set of compiler optimization passes to best reflect and utilize the underlying hardware.

Thesis Supervisor: Srinivas Devadas
Title: Professor

Acknowledgments

This work was done as part of the Execution Migration Machine (EM^2) project at the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL) under the supervision of Professor Srinivas Devadas. Its aim was to investigate the unusual cost model associated with compiling to a stack-based many-core architecture with a shared memory abstraction provided by migration of execution contexts, and to produce a C compiler targeting this architecture. I would like to thank my advisor for his patience, insight, and support throughout my time so far at MIT. I also would like to acknowledge my family, and all those close to me without whom my work would not have been possible. Finally, I would like to thank the computation structures group at the MIT CSAIL lab for copious feedback, ideas, and and insightful discussions that contributed to this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis aims to define a useful programming model for an execution migration platform (specifically EM^2, the Execution Migration Machine [36]) by exposing migration as a set of interfaces usable with the C programming language via a custom optimizing compiler. We employ hardware-software co-design to produce an EM^2 core architecture which serves to simplify the compiler problem and improve compiler efficiency. We also consider the ISA encoding in abstract terms to establish a baseline comparison of encoded instruction density to an ideal upper bound. Due to the unorthodox architecture of EM^2, no compiler well-suited for the platform exists at this time. Moreover, the compiler problem, although well studied for the type of architectures used by the majority of modern computers, is relatively unexplored in the context of both hardware stack-based architectures, and fine-grained thread migration. The platform offers a new cost model significantly different from that of common register-based out-of-order superscalar architectures, which gives us an opportunity to reevaluate the trade-offs associated with compilation for these architectures, and to explore novel algorithms, or novel applications of existing optimizations.

We carefully design the compiler backend, a library of EM^2-optimized parallel primitives, and a set of compiler optimization passes to best reflect and utilize the underlying hardware. Throughout this work, we attempt to gain a deep understanding of the costs and enabling features of the EM^2 platform by aggressive design space exploration. We use the insight gained to better inform the algorithms and heuristics used by the compiler, and propose a set of optimizations to improve utility of fine-grained thread migration by reducing frivolous migration due to eviction by judiciously ordering memory accesses and computation. This thesis is organized as follows: Chapter 2 sets up the framework to ground this work in reality, namely a compiler infrastructure based around the LLVM [34] framework. Chapter 3 discusses the hardware stack and the compiler design decisions and algorithms made to efficiently target this unorthodox architecture. Chapter 4 investigates encoding schemes for EM^2 instruction set by discussing the problem in abstract terms, deriving an upper bound on instruction density, and evaluating several practical schemes against this ideal. Chapter 5 considers library primitives for synchronization as an illustrative example of optimized primitives inexpressible above the abstraction imposed by the ABI, and explores execution migration as an enabling technology for synchronization. Chapter 6 introduces several compiler optimizations to improve performance and efficiency of programs by taking advantage of judicious fine-grained thread migration, partial context migration, and improving code density. Finally, Chapter 7 compares performance and efficiency of compiled code to equivalent hand-optimized code, and discusses limitations of our compiler, and future work.

1.1 Background Information

This work is relies on and serves to extend numerous existing projects. In this section, we summarize background information and related work to better provide context for the information in subsequent chapters.

1.1.1 Fine-Grained Thread Migration

Although thread and process migration has been a common feature provided by an operating system (OS), fine-grained migration (millisecond) is out of reach for a software service due to the latencies involved with a context switch to the OS. Migration at very fine intervals can be beneficial for power efficiency and performance, however, so several projects [46, 9, 7] used the idea for improvements ranging from better hardware utilization to performance gains to dark silicon. The Execution Migration Machine (EM^2) project views fine-grained thread migration as an enabling technology [35], and because applications are limited by migration granularity, seeks to reduce migration latency and bandwidth overhead to its lowest. EM^2 is a many-core architecture of 110 stack cores, shown in Figure 1-1, employing fine-grained thread migration to implement a scalable shared memory abstraction with no replication of data in its caches. The architecture also implements a remote access memory access protocol. In order to minimize average and minimum context size for migration, the EM^2 platform allows partial context migration, and to this end uses a stack-based core instead of a register-based architecture. Because a stack maintains a notion of temporal locality, a partial migration carries with it the values needed for the most immediate computation (although this places the burden of high quality stack scheduling on the compiler). Its purely hardware-based implementation can accomplish a single thread migration in as few as 4 cycles between the adjacent cores when a minimally-sized context is migrated, and 33 cycles between the most distant cores (on the 110-core grid) with a maximally-sized context. The cost of remote accesses is strictly lower. The architecture also implements a hardware migration predictor [51] to dynamically select between migration-based memory accesses and remote access via reinforcement learning, as well as selecting the appropriate context size for a migration.

1.1.2 Compilers

A compiler [3] is a collection of transformations that convert a program expressed in one language (source) into its implementation in another language (target). In general, the source is expressed in a high-level language such as C, while the target is architecture-specific, such as the language directly implemented by a computer system such as an OS over a particular processor architecture. In this case, the compiler targets an application binary interface (ABI): a union of the instruction set architecture (ISA, the language component of an ABI), a library of primitives (such as system calls), and a set of conventions that dictate how high-level constructs such

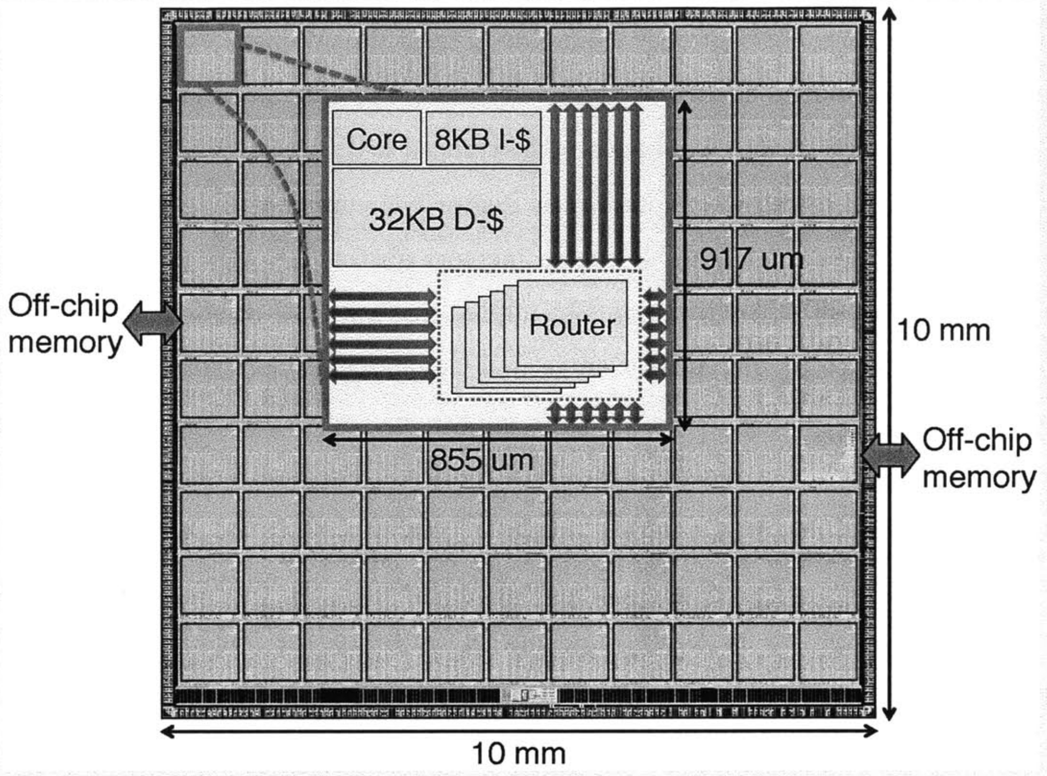


Figure 1-1: EM^2 chip architecture and layout

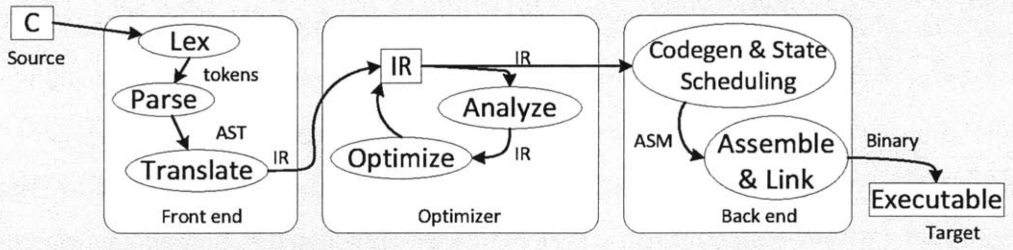


Figure 1-2: High-level architecture of a typical extensible compiler

as function calls are mapped to instruction sequences. The ABI defines an abstract environment within which a program executes - not on bare metal, but instead bound by a set of rules, and aided by a library. A compiler is a large and complex work of infrastructure (exemplified in Figure 1-2, so it seldom makes sense to create one from scratch. Instead, research projects rely on existing compiler infrastructure such as GCC [55], LLVM [34], and LCC [19] to provide the majority of algorithms and data structures, and allowing the project to focus on its goals instead of implementing complex infrastructure.

Compilation in the context of execution migration is a relatively new area of research. Although execution migration has long been considered as an OS-level operation moving a thread or process between cores [12], processors, or even between servers in a large-scale supercomputer [27], low-latency fine-grained migration has only recently been a subject of discussion in compiler research. Compilation for a stack architecture is also somewhat under-researched. Although the topic is quite old, it has only been seldom discussed. As Phil Koopman (a principal investigator in this area) stated in 1993, "Stack compilers aren't currently very efficient - but that's because no-one has tried very hard" [30]; the situation had not changed significantly in the last 20 years. The compiler problem problem in general, and register allocation specifically, is dissimilar from compilation to a register-based target: the in-core value store of a stack machine is logically infinite but not freely addressable, just the opposite of a register file. As a result, the cost model for a stack machine is largely unexplored, and may merit new algorithm choices at every level of the compiler flow. Several investigators including Koopman [32], Maierhofer [38], Bailey [6], and Shanon [50] made significant advances in understanding the stack cost model, and studied approaches to stack scheduling (equivalent of register allocation for a stack machine). In recent years, the main example of a hardware stack-based processor has been the UFO project [45], which attempts to show that stack-based architectures are not inherently less efficient than register-based machines by implementing complex out of order superscalar machines with hardware stack architected state, as well as a compiler, libraries, and other software.

1.2 Benchmarks

To inform compiler optimizations and measurements, we use a set of benchmarks, summarized in Table 1.1. Due to the absence of standard libraries designed for the EM^2 platform, we use simple benchmarks written in architecture-agnostic C code, and write several of our own. The benchmarks are gathered from various sources: FSU [10], MiBench [1], and several written specifically for this project. All benchmarks are evaluated on a bit-accurate ISA simulator emulating the underlying machine at the instruction level, with high-level features added where necessary to simulate partially compiled code.

Benchmark	Source	Description
`bsort`	FSU	A binary sort benchmark
`image`	FSU	Image smoothing, based on a FSU benchmark
`matmul`	FSU	A benchmark comparing several matrix multiplication algorithms
`fib`	(custom)	Iterative implementation of Fibonacci
`life`	FSU	Conway's life simulation
`texture`	(custom)	A single-thread implementation of the `synthesis` (Section 1.2.2) benchmark
`qsort`	FSU	An implementation of an integer quicksort algorithm
`queens`	FSU	A solver for the eight queens problem
`towers`	FSU	A solver for the towers of Hanoi problem
`bitcount`	MiBench	Bit counting benchmark
`regex`	(custom)	An implementation of a regular expression matching algorithm
`diff`	(custom)	A customized implementation of string comparison based on the `diff` program
`fact`	FSU	Recursive implementation of Fibonacci

Table 1.1: Single-threaded benchmarks used in this work

1.2.1 Microbenchmark Workloads

Chapters 5 and 6 implement several micro benchmarks to evaluate specific aspects. The microbenchmark algorithms are given in the relevant sections, and are expressed at a very low-level EM^2 (exposed via C function headers to interact with compiled code). To evaluate the microbenchmarks, we simulate the RTL of a 110-core Execution Migration Machine (specifically, the EM^2 chip), programmed via loading the benchmark binaries into the appropriate instruction caches. A simple fixed-latency memory interface was used to model interaction with DRAM, which does not significantly affect microbenchmark measurements because the data sets used fit in on-chip caches.

1.2.2 Parallel Benchmarks

Some aspects of this thesis deal with optimizing the interaction between threads running on multiple cores of the Execution Migration Machine, so single-threaded benchmarks are not adequate to demonstrate the behavior associated with parallel workloads: simple benchmarks have no inter-thread communication, no synchronization, and no migration. To study the behavior of parallel workloads, we compile a set of four parallel benchmarks each utilizing multiple cores on the EM^2 platform. We describe all benchmarks in generic C code and use the ABI library for synchronization and other tasks, where applicable. All benchmarks are borrowed from previous work on the EM^2 project [36], and evaluated on a bit-accurate ISA simulator emulating the underlying machine at the instruction level.

Table Scan

The table scan benchmark, `tscan`, is a single-threaded workload with a large data set sprawled among many cores in the system. The benchmark is designed to demonstrate the cache aggregation effect whereby various parts of the data set are cached on their corresponding home cores, with the thread migrating among the participating cores to access the data, effectively utilizing a large collective cache: the union of all caches of participating cores. For the purposes of this work, we benchmark a table scan algorithm touching a data set of 128 entries per core, spanning 32 cores. Our ISA simulator does not model the memory hierarchy in detail, so we simply examine the migration behavior induced by this benchmark.

Cross-Validation

The cross-validation benchmark, `xvalidate`, evaluates a simplified linear regression model over a training data set, and tests the model against a test set. Because the choice of data and test sets have a significant effect on the results, cross validation models and tests the models using several different subsets of the data. The `xvalidate` benchmark uses N threads each having local access to one of N partitions of the data set. Each thread builds a linear regression model using $N-1$ partitions of the data set, and evaluates the model against the remaining partition. No two threads test against the same partition. By employing a barrier, we orchestrate the migration behavior in the system, ensuring no evictions due to core contention. Partial context migration is key to avoid excessive evictions: threads easily make forward progress on the guest core, where their current data set partition is, meaning run length is very important in this benchmark. For the purposes of this work, we implement cross-validation as a linear regression of 32 data set partitions with 128 values in each data set partition.

Parallel Prefix Sum

The parallel prefix sum benchmark, `ppsum`, is a parallel benchmark whereby a collection of threads reduces a data set to a set of partial sums of the same cardinality. Figure 1-3 illustrates this operation, and shows how the benchmark evaluates this operation in parallel. For the purposes of this work, we implement parallel prefix sum using 32 threads, each of which is local to a data set partition of 128 values, meaning the benchmark executes 12 iterations, blocking on a barrier between each iteration.

Example-Based Texture Synthesis

The example-based texture synthesis benchmark [18], `synthesis`, is a set of loops that search an "example" image (a 2d array of integers) for the best-suited pixel (value) to insert into a synthesized image. In simple terms, it attempts to probabilistically fill

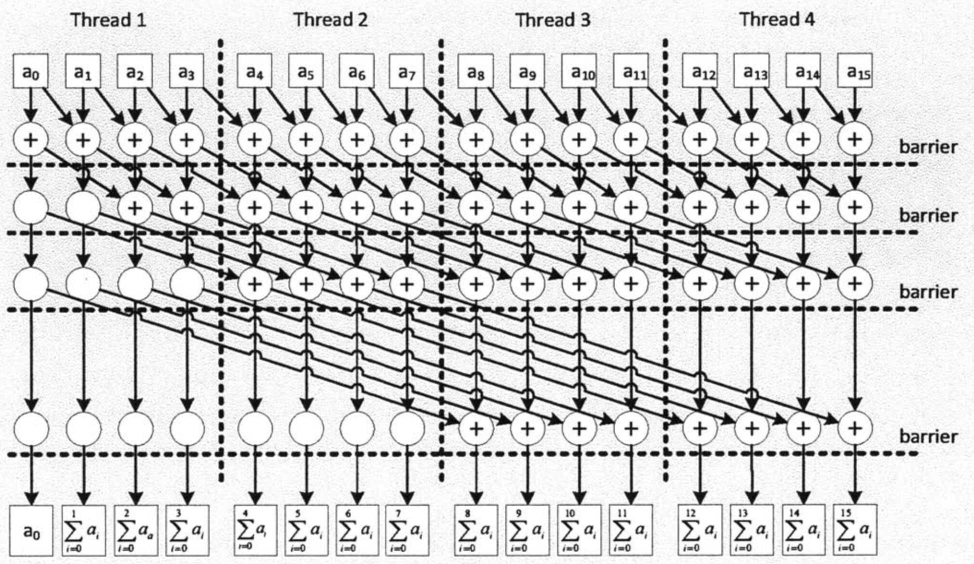


Figure 1-3: Illustration of a parallel prefix sum DFG and thread partitioning

out a bitmap with the same statistics as an example bitmap, for instance extending repeating textures.

This technique is useful for synthesis of non-repeating textures (shown in Figure 1-4, removal of objects from photographs, image morphing, and super resolution in video processing. Using a multiresolution algorithm [59] allows us to search for multiple pixels in parallel. For the purposes of this work, we implement parallel prefix sum using 32 threads, each of which is local to a portion of 8×8 pixels of the example image, 16×16 pixels of a low-resolution prior, and a 32×32 values of the synthesized texture. We run the algorithm for 10 iterations, synthesizing a total of 320 pixels.

1.2.3 Manually Optimized Benchmarks

Due to there being no other compilers available for the EM^2 platform, or even for similar architectures, we are unable to conduct a comparative study to evaluate the relative efficiency of the compiler described in this thesis. Although quantifying the absolute efficiency of a compiler is a difficult and imprecise task (the evaluation de-

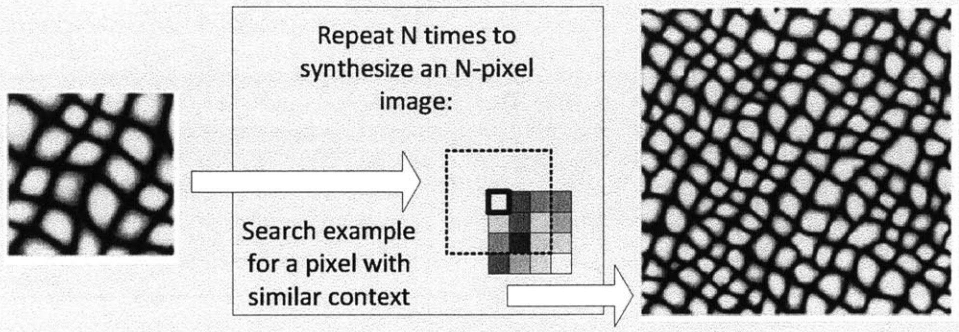

Repeat N times to synthesize an N-pixel image:
Search example for a pixel with similar context

Figure 1-4: Illustration of example-based texture synthesis

pends heavily on quality of implementation of both the baseline and the source C code equivalent), we implement three of the parallel benchmarks, specifically **`tscan`**, **`xvalidate`**, and **`ppscan`**. In Chapter 7, we compare the performance and efficiency of the compiler in context of these three workloads, and use the comparison to inform a discussion of the various overheads associated with the EM^2 compiler.

Chapter 2

EM^2 Core ISA as a Compiler Target

An instruction set architecture is the language component of an application binary interface (ABI, which also encompasses a library and the set of conventions that describe high-level features of programs written for the ABI). Although an instruction set is a language that is directly interpreted by the underlying hardware, the writer of the language is seldom human. Instead, most programs are compiled from source code written in productive high-level languages such as C. In general, the ISA is built to meticulously describe hardware capabilities, while users rely on a complex custom compiler backend for translation between a compiler's intermediate representation and the target machine's ISA. Despite the high expressive power of an ISA designed this way, the compiler backend for a highly architecture-specific instruction set is a very complex and inexact tool: the compiler must translate a program between languages with different fundamental assumptions and goals, often making the translation process very open-ended, with many conflicting optimization goals. As a result, a compiler backend targeting a highly specialized ISA is difficult to build, makes conservative assumptions about the code it compiles, and often introduces overhead in form of performance bugs and workarounds for conflicting semantics between the source (IR) and target (ISA) languages. A thorough peephole pass is only able to fix some of the most localized inefficient instruction sequences.

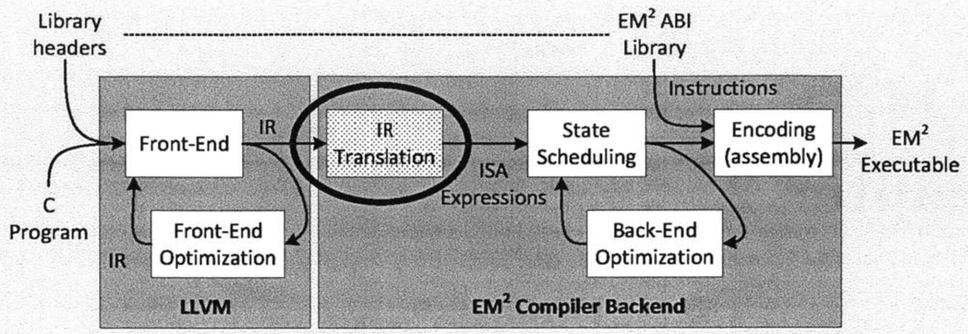


Figure 2-1: High-level role of ISA translation in the EM^2 compiler

This chapter describes the framework we use to ground the entire discussion of the EM^2 compiler backend: we choose the LLVM compiler project to extend with a large set of EM^2-specific transformations and optimizations to implement a complete compiler. We begin this discussion by describing the EM^2 core instruction set, which is compiler-efficient by design. We designate the compiler's intermediate representation a first-class constraint, and derive the ISA by implementing simple and elegant compiler backend translation. A high-level view of the compiler backend, and specifically the role of ISA translation in context of the compiler backend is shown in Figure 2-1. Although to produce executable code we must solve several additional problems (the subject of subsequent chapters), this approach ensures that the outstanding subproblems are well-defined, and isolated from other compiler transformations. By designing the ISA to be maximally compatible with the compiler's internal representation, we avoid having to figure out a complex translation between two languages with different fundamental assumptions. Even though this chapter addresses a largely *engineering effort* (a fair comparative study between different ISAs is difficult to imagine), we observe that the resulting set of instructions closely resembles a typical RISC ISA, as expected for a compiler-centric instruction set. We argue that any lost opportunity at making the ISA more dense or more expressive as a result of this hardware-software co-design is overshadowed by an efficient, simple, and modular compiler backend requiring relatively little effort to implement, allowing us to invest additional effort in compiler optimizations.

2.1 Choice of Compiler Framework

When developing a compiler for a custom platform, it is generally not advisable to build the entire system from scratch. A compiler is a complex and high-value project with sizable upfront development costs, with only a subset of the work required pertaining specifically to the architecture the compiler is designed to target. Considering the difficulty involved in verifying a compiler's correctness, it makes sense to avoid developing excessive new (and therefore untested) code.

Fortunately, there are numerous existing compiler projects, such as GCC [55], LLVM [34], and LCC [19], to name a few, designed to target multiple platforms, all of which offer examples and community experience, making the task of designing a custom backend more manageable. We found the LLVM compiler infrastructure project to be well-suited as a starting point for the Execution Migration Machine compiler because of its modular infrastructure, favorable license [42], and a diverse set of existing compiler backends [2].

The LLVM compiler is built around a powerful low-level language (The IR [33], or intermediate representation), which attempts to describe computation in an machine-agnostic way. Specifically, it maintains a rich set of data types [33], which allows the backend much flexibility in implementing operators over complex data types in an efficient manner on the target machine. The LLVM language also makes minimal assumptions about the target hardware, a good fit for the EM^2 project's unorthodox stack-based architecture.

2.2 Compiler IR Translation to ISA Expressions

To avoid designing a separate language, which may complicate the compiler backend by forcing a translation between two dissimilar languages, we enumerate an "executable" subset of the LLVM intermediate representation - namely the instructions that, given proper scheduling of machine state, can be executed on the EM^2 core datapaths directly. In order to do this, we replace non-executable operators with one or more "executable" operators, or a procedure call. These transformations are independent of one another, and can be performed in any order. Program structure is unaffected. A build script environment [58] ties this set of transformations along with other transformations detailed throughout this thesis into a complete compiler backend. In this section, we discuss the transformations of major families of operators in the LLVM IR into expressions (one or a sequence of "executable" instructions), the elements of which are directly implemented in the EM^2 ISA. This practice of hardware-software co-design in enumerating the core ISA allows the compiler backend to remain modular and easy to verify[1]. The section also discusses data type conversion, where applicable. Finally, we present a table listing the LLVM instruction set 2.1, and how each instruction family maps to the executable subset.

2.2.1 Unsupported language features

The LLVM ISA is powerful, and contains many features for compatibility with major architectures and languages. Some operators never arise when compiling generic C code, and the EM^2 compiler treats these features as unsupported. For example, architecture-specific types such as the `x86mmx` (MMX [15] type for an x86 architecture) is unsupported. Metadata types (program annotations) are ignored in the backend, but are used aggressively during program optimization. Closures and low-level exception operators are unsupported, as they assume implicit high-level state, and are not used when compiling languages with straightforward control flow such as C. Types with no well-defined size (void, unknown types) are unsupported as well, unless resolved into sized types such as pointers.

[1]A compiler is never verifiable because it is stateful and does not produce a fixed-length output, so we refer here to the process of verifying a compiler's output by understanding the compiler mechanism, and sufficiently testing its implementation.

2.2.2 Arithmetic and Bitwise operators

Most math operators are implemented directly, as long as they operate on integer types that can be represented as a 32-bit two's complement sequence. Operators over larger integers are broken into multiple operations on smaller integers. Conversion operators are omitted (no-op) for conversions between types that can be represented as a 32-bit two's complement sequence, else appropriate conversion sequences are implemented. Wide types are represented as multiple 32-bit words: $\alpha_{64} = \alpha_{hi}*2^{32}+\alpha_{lo}$ for any arbitrary variable α.

Bitwise operators over wide integer types are trivially emulated using 32-bit operators. Observe that $a_{64}\&b_{64} = (a_{hi}*2^{32}+a_{lo})\&(b_{hi}*2^{32}+b_{lo}) = (a_{hi}+b_{hi})*2^{32}\&(a_{lo}+b_{lo})$. Therefore, the sequence $a_{lo}\&b_{lo}; a_{hi}\&b_{hi}$ correctly implements the and operator for a 64-bit type.

Addition is more complex, as it requires a carry propagation. Still, $a + b = (a_{hi} * 2^{32} + a_{lo}) + (b_{hi} * 2^{32} + b_{lo}) = (a_{hi} + b_{hi}) * 2^{32} + (a_{lo} + b_{lo})$, so the algorithm for a 64-bit addition is simple:

```
Let $sum_{lo} = a_{lo} + b_{lo}$;
Let $c$ = carry bit from the $sum_{lo}$ computation.
Let $sum_{hi} = a_{hi} + b_{hi} + c$;
```

Multiplicative product for wide types is derived in much the same way: $a * b = (a_{hi}*2^{32}+a_{lo})*(b_{hi}*2^{32}+b_{lo}) = (a_{hi}*b_{hi})*2^{64}+(a_{hi}*b_{lo})*2^{32}+(a_{lo}*b_{hi})*2^{32}+(a_{lo}*b_{lo})$. The carry computation is somewhat involved for computing a wide product.

The front-end compiler does not produce arithmetic operators over types wider than 64 bits when compiling generic C code.

Complex function units in out-of-order superscalar cores improve performance by 1) using a hardwired datapath to perform the function, which is more efficient than a software emulation, and 2) allowing integer computation to proceed in parallel (with proper compiler provisions) while the multi-cycle function unit is busy. It is important to note these units typically have low utilization. The EM^2 architecture is a chip multicore with over 100 datapaths and features to make thread creation extremely inexpensive. The machine provides a lot of parallelism by virtue of having a large number of lightweight threads of execution. The advantage of hard-wired datapaths has less benefit on EM^2 relative to high performance single-threaded architectures. We therefore choose to emulate complex operators using a software library call for the purposes of this project. Specifically, we omit logic to compute a quotient and remainder, as well as a floating point unit.

Division and remainder are implemented using a high-radix algorithm (radix 4) [41] to reduce the carry length. The algorithm is written in C for all bit widths used in the program; divide and remainder instructions are replaced with a procedure call to the generated code. The algorithm for computing a remainder is given in Figure 2-2. Division is similar, but must also maintain a quotient variable. Although the division algorithm suffers from additional cycles needed to orchestrate variables on the stack, and serializing conditional statements, it is not dramatically slower than a hardware divider without requiring any additional hardware (a hardware divider is a very expensive and high-latency function unit). An existing high-quality library [23] implements floating point in portable C.

2.2.3 Comparison operators and Control Flow

Although many architectures [28, 14] combine control flow instructions with comparison instructions (after all, control flow is often conditional on a comparison), the LLVM IR maintains the two as separate operators. This is useful for architectures [54] that implement comparison operators explicitly. We implement the LLVM comparison operators directly, as we have done with arithmetic operators, to simplify the backend translation pass. This way, conditional control flow operators are conditioned on a single boolean argument (booleans are encoded as 32-bit integer, as in the K&R [29] C specification), and have a single explicit jump target, the next instruction being the implicit target if the condition is false.

Conditional control flow is makes up a rich set of operators in LLVM, including a `switch` operator with an unbounded number of targets. Because all are trivially reducible to conditional branches and jumps, as shown in Figure 2-3, we include

```
static unsigned short remainder(
    unsigned short divisor,
    unsigned short dividend,
    unsigned short bit_width) {
  unsigned char i;

  if(dividend == 0){
    unreachable();
  }
  i = bit_width;
  while(i > 0){
    unsigned int d1,d2,d3;
    i -= 2;
    d1 = dividend << i;
    d2 = dividend << (i+1);
    d3 = d1+d2;

    if(divisor < d1)
      // nothing
    else if(divisor < d2)
      divisor -= d1;
    else if(divisor < d3)
      divisor -= d2;
    else
      divisor -= d3;
  }
  return divisor; // remainder
}
```

Figure 2-2: Source code for a radix-4 remainder procedure

only these two types of control flow in the "executable" subset of the IR. Multi-target `switch` statements are rewritten as a sequence of branches by an optimization pass before the backend is invoked. Indirect branches (`indirectbr`) are essentially a jump, but the IR enumerates all possible branch targets, allowing variables to be better scheduled across basic blocks (as discussed in Chapter 3).

A `call` instruction is a jump, but deposits the return address onto the stack. We do not implement a `return` instruction, as it is semantically identical to a `jump`.

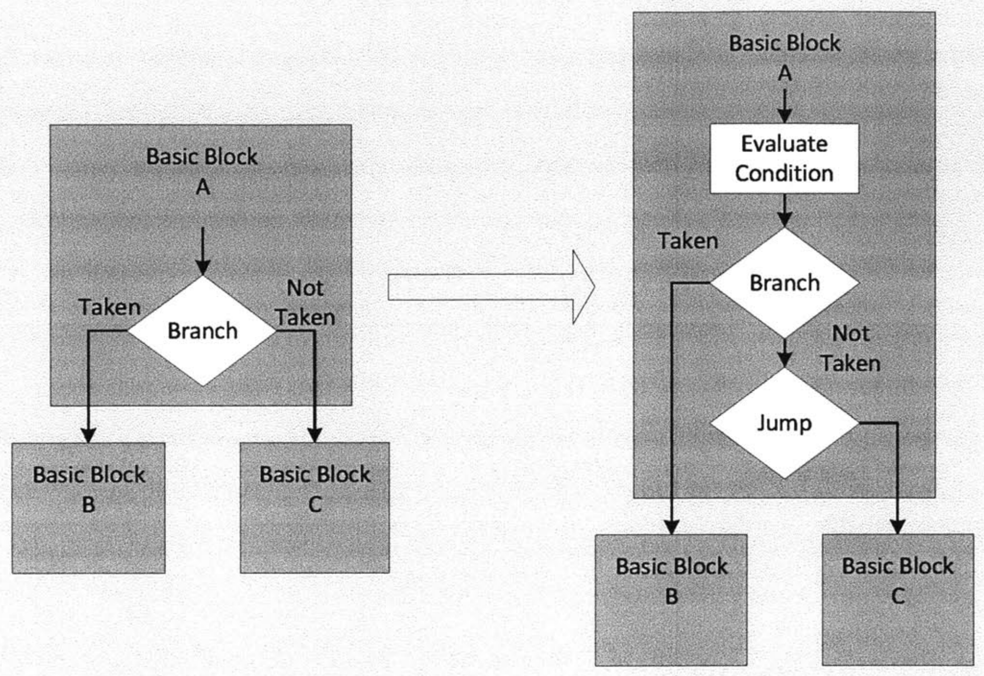


Figure 2-3: Transformation from IR branch to EM^2 single-target control flow

Calling convention is used to pass return variables (discussed in detail in Chapter 5). Wherever possible, we emit position independent code by using `branch`, `jump`, and `call` offsets, as this eliminates additional instructions needed to set up the `branch` or `jump` target address, and allows for better compression of encoded instructions (more about compressed encoding in Chapter 4). Labels are preserved until very late in the backend, when the executable is stitched together from optimized and encoded code segments. Although this chapter does not discuss the ISA encoding, it is important to note that any choice of encoding would limit the maximum range of an offset, affecting overall efficiency of the ISA.

2.2.4 The Phi operator

The `phi` operator is a special primitive in the IR used to handle ambiguously defined variables. The LLVM language presents all variables in single static assignment (SSA) form, making analysis simple [17], but this representation does not quite allow a full range of programs to be written with the flexibility required for a useful intermediate representation. The `phi` operator extends SSA by implementing a multiplexer, or a merge operator [8], effectively renaming one or more variables from a basic block earlier in the control flow graph, and creating a new variable to hold the selected value. The operator does not imply any computation, and is statically elaborated in the EM^2 compiler backend by tracking dependencies across basic blocks.

2.2.5 Data Type Conversions

As shown throughout this chapter, we translate operations over wide and complex types into equivalent sets of operations over types the EM^2 architecture can work with directly. In other words, an IR variable is translated into one or more expression variables, meaning there exists a surjective mapping from IR variables to expression variables. The most complex data type conversions are wide integer and floating point, which are discussed in Section 2.2.2. Smaller integer types are trivially converted to 32-bit integers, although overflow checks are appropriate in some cases. There is no consensus with regard to correct behavior on overflow among C implementations, so we take the path of least resistance and ignore overflow conditions in most cases, except when truncating the variable to a correct bit width.

Pointers and function types are implemented as 32-bit integer addresses into the global memory, which is consistent with the K&R [29] C specification. The pointer type is derived from the type it represents, which allows us to correctly perform memory accesses in case of pointers to large types. The IR has rich compound types such as vectors, arrays, and `structs`, all of which we implement. In the case of wide derived types (member, or contained types), the components of the vector, array, or structure are aligned to 32-bit boundaries. The executable subset of the IR does not have operators for these types, so we split them into primitive types that fit into 32-bit integers, much like we do with wide types.

The IR heavily relies on the `getelementptr` operator, which encompasses all address math used to determine offsets in nearly all accesses to complex data structures. The getelementptr operator does not describe any dynamic computation, so we statically interpret as an address lookup or sub-type selection, selecting a primitive variable in a compound type. In limited cases, the much less powerful operators insertvalue, extractvalue, insertelement, extractelement, and shufflevector occur in the IR, which we implement by simply referencing the appropriate variable within the compound type.

The EM^2 compiler backend never packs small data types into 32-bit integers because unpacking costs extra computation, and therefore extra cycles. We do not attempt to argue this approach from an efficiency point of view. One side effect is internal fragmentation of memory words if an abundance of small types are used. For example, byte-character strings are very sparse in memory.

2.2.6 Constant Expressions

The LLVM IR treats constants as variables with a known value. For example, in a given program, all constants with a value (1_{32}) are denoted by the same 32-bit variable, defined to have the value of 1. We add two special instructions to the ISA to handle constants: push and set_hi, which together allow us to create integer constants. The compiler backend emits special constant expressions that simply output the constant value. These expressions do not require any special treatment, and can be scheduled with all other expressions. Aggressive constant folding by the LLVM optimizer avoids unnecessarily complex constant expressions. An optimization replicating the constant expression at all points of use instead of passing its value between uses is described in Chapter 6.

2.3 Evaluation of IR Translation Overhead

The core ISA is summarized in Table 2.2. Although there are a number of special instructions omitted, mostly pertaining to the hardware stack and system management eccentricities of the EM^2 architecture, the core ISA is essentially a subset of the LLVM IR. The translation mechanism is a series of trivial and independent graph rewrite passes, as detailed Table 2.1, so translation overhead is low.

LLVM Instruction(s)	Implementation in EM^2 Expressions
and, or, and xor	Bitwise arithmetic operators are implemented directly for types of 32 bits or smaller (using and, or, b_not and xor), and are emulated via a library procedure call for wide integer types.
shl, lshr, and ashr	Bit shift operators are implemented directly for types of 32 bits or smaller (using sill, srl and sra), and are emulated via a library procedure call for wide integer types.
add, sub, and mul	Simple arithmetic operators are implemented directly for integer types (using add, sub and mul), and are emulated via a library procedure call for wide integer types.
udiv, urem, sdiv, and srem	Integer division and remainder operators are emulated via a library procedure call.
icmp	Integer comparison is implemented directly via the following EM^2 instructions: comp_eq, comp_ne (equality); comp_ugt, comp_uge, comp_ult, comp_ule (unsigned comparison); comp_sgt, comp_sge, comp_slt, and comp_sle (signed comparison).
fadd, fsub, fmul, fdiv, and frem	The EM^2 architecture does not implement a floating point unit, so all floating point arithmetic is emulated via a library procedure call.
fcmp	Floating point comparison is emulated via a library procedure call.
	(Continued on next page)

Table 2.1: LLVM instruction families, equivalent in EM^2 Instructions

Table 2.1 – continued from previous page

LLVM Instruction(s)	Implementation in EM^2 Expressions
`fptrunc..to`, `fpext..to`, `fptoui..to`, `fptosi..to`, `uitofp..to`, and `sitofp..to`	Conversions to, from, and between floating point types is emulated via a library procedure call.
`call`	Procedure calls are directly implemented using `call` for absolute addresses, and `call_pc` for position independent code.
`ret`	Procedure call returns are implemented using a jump (`j`); return value is passed according to the EM^2 calling convention.
`phi`	NO-OP: phi operators are statically resolved, as discussed in Section 2.2.4.
`va_arg`	NO-OP: No special instructions are needed to implement variable argument lists because they are trivially implemented according to the EM^2 calling convention.
`br`, `switch`, `select`, and `indirectbr`	All control flow is reduced to conditional branches and unconditional jumps. Conditional branches are implemented using branches (`b_z` and `b_nz`) and jumps (`j`, `j_pc`), as shown in Figure 2-3. IR jumps are implemented directly.
`unreachable`	Not implemented: IR does not specify the behavior of this operator.

(Continued on next page)

Table 2.1: LLVM instruction families, equivalent in EM^2 Instructions

Table 2.1 – continued from previous page

LLVM Instruction(s)	Implementation in EM^2 Expressions
`extractelement`, `extractvalue`, `insertelement`, `insertvalue`, and `shufflevector`	NO-OP: all static selections of sub-types within a complex type are statically resolved by the compiler.
`alloca`	NO-OP: as later shown in Chapter 6, the EM^2 compiler resolves all static allocations at compile time by creating hardware stack variables.
`getelementptr`	NO-OP: this operator is a selection of a sub-type within a complex type, and is resolved statically by the compiler.
`load` and `store`	Implemented directly (using `fnc_ld` and `fnc_st`) for types that fit into a 32-bit word. Otherwise broken into a sequence of adjacent loads and stores. EM^2 also provides additional instructions (`ld_em`, `ld_ra`, `ld_rsv`, `ld`, `fnc_ld_em`, `fnc_ld_ra`, `fnc_ld_rsv_em`, `st_noack`, `st_em_noack`, `st_ra_noack`, `st_cond`, `st_em`, `st_ra`, `st`, `fnc_st_em`, and `fnc_st_ra`) to specify low-level behavior; these may violate memory consistency and are not emitted by the compiler, although these instructions are used in controlled ways by the ABI library, as shown in Chapter 5.
`trunc..to`, `zext..to`, and `sext..to`	Trivially implemented (NO-OP) for small integer types, otherwise emulated via a library procedure call.
	(Continued on next page)

Table 2.1: LLVM instruction families, equivalent in EM^2 Instructions

Table 2.1 – continued from previous page

LLVM Instruction(s)	Implementation in EM^2 Expressions
`ptrtoint..to`, `intoptr..to`, and `bitcast..to`	NO-OP: all types are translated into bit fields by the compiler, so bit conversion operators are trivialized. Pointers are implemented as integers in EM^2.
`fence`, `cmpxchg`, and `atomicrmw`	Not implemented: the front-end compiler never emits atomics when compiling architecture-agnostic C. EM^2 provides library primitives for synchronization, which use atomics directly, as discussed in Chapter 5.
`invoke`, `resume`, and `landingpad`	Not implemented: the front-end compiler never emits closures or hardware-supported exceptions when compiling C.
intrinsic functions	Not implemented: intrinsics that remain to be elaborated at the compiler backend are architecture-specific, and EM^2 does not define any intrinsic functions.
constant functions	Constant functions are statically resolved by the compiler back-end.

Table 2.1: LLVM instruction families, equivalent in EM^2 Instructions

This chapter details an engineering effort in deriving a compiler-friendly instruction set, but does not attempt to measure the quality of the resulting ISA relative to other instruction sets. Indeed, a fair comparative study between instruction sets is difficult to imagine because ISA design decisions carry far-reaching implications such as changes in implementation patterns, code density, instruction cache behavior, decoder complexity, pipeline latency and utilization, and other factors that all affect overall performance. The core ISA we derived from the IR is very similar to a typical RISC ISA, but with significant potential for compression, as explored in Chapter 4. Instead of arguing the quality of our ISA, we observe that this exercise in hardware-

EM^2 Instruction(s)	High-Level Function
push and set_hi	Constant values
and, or, xor, and b_not	Bitwise arithmetic
sll, srl, and sra	Bit shift operators
add, sub, and mul	Two's complement integer arithmetic
comp_eq, comp_ne, comp_ugt, comp_uge, comp_ult, comp_ule, comp_sgt, comp_sge, comp_slt, and comp_sle	Signed and unsigned integer boolean comparison
fnc_ld and fnc_st	Simple, blocking memory access. Defers decision between remote access and migration to hardware predictor. Compiler emits these instructions wherever no obvious migration can be inferred
fnc_ld_em, fnc_ld_ra, fnc_st_em and fnc_st_em	Blocking memory access that specify remote access or migration
fnc_st_em_noack, fnc_st_em_noack, fnc_st_ra_noack st, st_em_noack, st_em, st_em_noack, st_ra, st_ra_noack andld	Non-blocking memory accesses (without fnc), and memory accesses generating no acknowledgment packet (noack). These may break memory consistency, and are used by the ABI library (which will be exemplified in Chapter 5) in highly controlled ways
fnc_ld_rsv, fnc_ld_rsv_em, fnc_ld_rsv_ra, ld_rsv, ld_rsv_em, ld_rsv_ra, st_cond, fnc_st_cond, fnc_st_cond_noack st_em_cond, fnc_st_em_cond, fnc_st_em_cond_noack st_ra_cond, fnc_st_ra_cond, and fnc_st_ra_cond_noack	Atomic memory instructions (based on the load-reserve and store conditional pattern). These and are used by the ABI library to implement synchronization primitives
call and call_pc	Procedure call
jump and jump_pc	Unconditional control transfer
b_z and b_nz	Conditional control transfer

Table 2.2: EM^2 core instructions

software co-design by reducing the ISA from the IR allowed us to quickly build most of a high-quality compiler backend. Although the backend described in this chapter is incomplete (state scheduling, encoding, optimization are not solved in this chapter, and are covered in Chapters 3, 4 and 6, respectively), these problems are well-defined and isolated from other aspects of the backend. This productive approach to designing a compiler backend allows us to invest significant effort in optimizing the result, arguably producing very good overall results, despite likely missed opportunities at a more expressive ISA.

To argue low translation overhead, we conduct an experiment: after compiling a set of representative benchmarks, we symbolically execute them on a simulator, and examine the traces to compute the overhead incurred in translating the LLVM IR into the executable subset. This overhead excludes the cost of scheduling variables onto architected state (a register file equivalent would be spilling and loading the stack frame into registers); we defer a discussion of stack scheduling overhead to Chapter 3. The overhead here is defined as $\text{Overhead} = \frac{\#\text{core instructions in compiled trace}}{\#\text{instructions in IR, excluding no-ops}}$. The results are shown in Figure 2-4, showing that control flow contributes most to translation overhead in nearly every benchmark. Benchmarks using division or wide types incur significant overhead due to expansion of arithmetic expressions, and memory-intensive benchmarks incur overhead due to address computation. On average, a single IR operator (excluding IR operators trivialized into no-ops) is translated into an expression of 1.108 EM^2 instructions.

2.4 Summary

In this chapter, we discussed a methodology by which we co-designed a partial compiler backend and its target ISA. The result is an efficient set of translation mechanisms that implement the rewrite rules of a compiler backend. We argue that the remaining components of the backend (explored in subsequent chapters) are well-defined and isolated problems, a property that allows the backend to be modular, easy to understand and optimize. The approach we take is productive and allows us

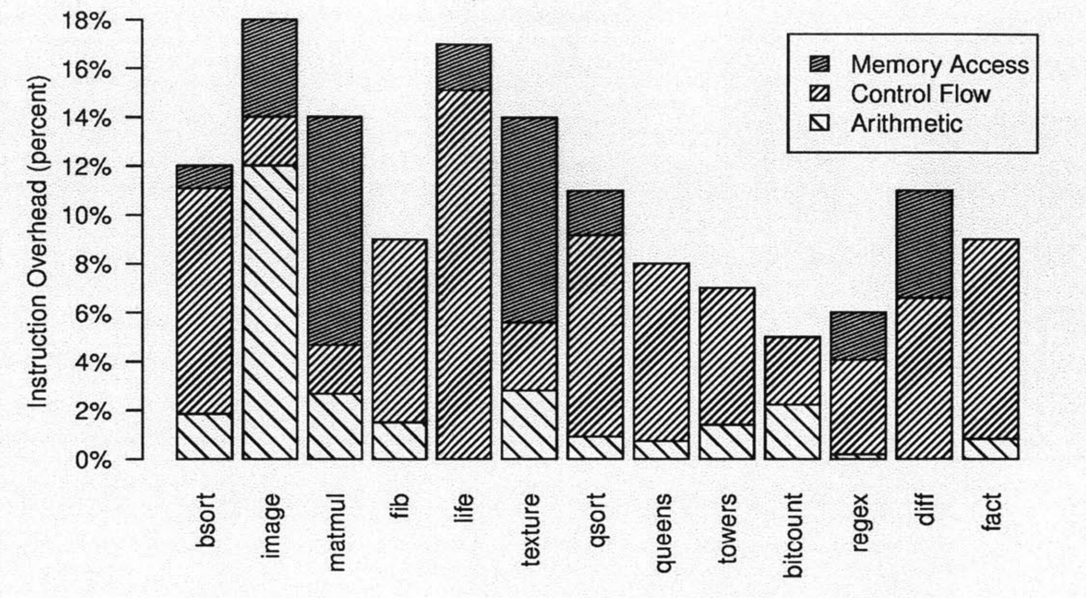


Figure 2-4: Overhead incurred translating IR to expressions of EM^2 instructions

to build a high-quality compiler in an academic setting with limited manpower.

Workloads for the Execution Migration Machine are given in C, so a high quality compiler can obtain high quality results despite an ISA with potential missed opportunities for added efficiency.

Chapter 3

Scheduling Program Variables onto the EM^2 Architected State

The core ISA instructions defined in Chapter 2 are sufficient to build an expression that emulates any IR operators emitted by the compiler front-end, assuming we compile architecture-agnostic (and single-threaded) program written in C. It is not, however enough to perform any real computation: the expressions operate on variables, which have only logical meaning, but not yet any specific associated state in the underlying hardware. The compiler backend must perform additional work to allocate and schedule hardware resources to variables used by any expression sequence. Furthermore, because EM^2 cores are stack machines with no register file, the backend must appropriately order all expressions to ensure all dependencies are met and the right variables are available at the top of the stack when an expression executes. Whenever no such ordering exists, the compiler must insert instructions to stage the stack by rearranging its entries. This is the problem of scheduling variables onto the EM^2 architected state. A high-level view of the compiler backend, and the role of variable scheduling in the larger context of a compiler backend is shown in Figure 3-1.

In this chapter, we will define the problem of scheduling expression variables on to the hardware stack, and propose a two part solution: optimizing expression order, and allocating state greedily by favoring least-cost-of-access. We formulate a local

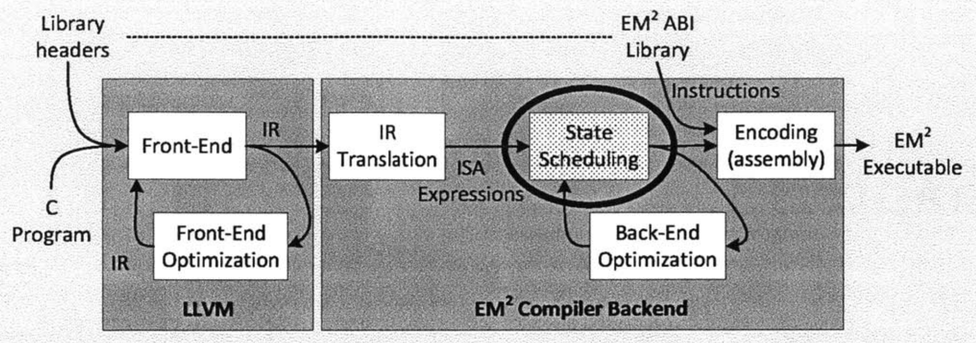


Figure 3-1: High-level role of variable scheduling in the EM^2 compiler

stack scheduling algorithm based on the well-studied Koopman's algorithm [32]. We also formulate an approximating cost-based search algorithm for stack reordering and a heuristic algorithm for scheduling variables across basic block boundaries. We argue the complexity of the proposed algorithms, and evaluate several variations using a range of benchmarks. Finally, we discuss the shortcomings of this scheme, and explore heuristics to improve our solution in various common scenarios.

3.1 Expression Variables and the Local Stack

The variables over which IR instructions perform computation are defined by a single static assignment (SSA) [17], followed by zero or more uses (other IR instructions using the variable as input). Core ISA expression variables follow the same pattern, as there is a surjective mapping to corresponding IR variables (multiple 32-bit expression variables may derive from a single IR expression if it is of a complex or otherwise wide type).

EM^2 has no register file, so all expressions take inputs from the top of the stack, and deposit outputs onto the top of the stack also. Chapter 2 explains how complex operators are implemented as expressions (sequences of core ISA instructions). For correctness, the expressions must be given in correct order and communicate internally via the hardware stack to pass temporary values between instructions. We collectively

EM^2 Instruction	Consumed	Produced
`push`	0	1
`set_hi`	1	1
`and`, `or`, `xor`, and `b_not`	1	1
`sll`, `srl`, and `sra`	1	1
`add`, `sub`, and `mul`	2	1
`comp_eq`, `comp_ne`, `comp_ugt`, `comp_uge`, `comp_ult`, `comp_ule`, `comp_sgt`, `comp_sge`, `comp_slt`, and `comp_sle`	2	1
(All load instructions)	1	1
(All store instructions)	2	0
`call`	1	1
`call_pc`	0	1
`jump`	1	0
`jump_pc`	0	1
`b_z` and `b_nz`	1	0

Table 3.1: EM^2 core instructions as producers and consumers of stack values

define these temporary, intermediate values the "E-stack" region (following existing work [49]), precisely the region used within an expression. Each expression takes a fixed, statically known number of inputs from the top of the hardware stack, and produces a fixed, statically known number of outputs, which are again placed on top of the stack. This means that given a trace of expressions, we can statically analyze the state of the hardware stack at any point in the trace by considering the producer and consumer behavior of each prior operator. Table 3.1 enumerates this producer and consumer behavior of all core ISA instructions, as defined in Chapter 2. Although the space of all expressions is not enumerable easily, stack requirements for any expression are trivially calculated by considering the stack behavior of each instruction in the expression.

Clearly, each expression requires its variables to be at the top of the hardware stack immediately prior to execution. In order to correctly execute an IR instruction sequence, we must maintain the following invariant by manipulating the ordering of the hardware stack as needed:

The top of the stack immediately prior to execution of an operator $F(i_1 \cdots, i_2 \cdots) \rightarrow \{o_1, o_2, \cdots\}$ must be of the form $(i_1, i_2, \cdots)$.

Stack manipulation is not derived from the program at a higher level, so we do not try to figure out how to get those instructions from the source program (any stack manipulation is strictly overhead, as it does no work relevant to the high-level program, much like spilling in a register file architecture). Instead, we emit additional instructions to manipulate the hardware stack as necessary prior to each expression, bringing the values it takes as input to the top of the stack.

3.2 Stack Access Depth

The stack is a data structure highly optimized for last-in-first-out (LIFO) access pattern, which is a good fit for many workloads since most variables tend to be short-lived, used only to communicate intermediate values between adjacent expressions. In some cases, however, the expression ordering fails to naturally maintain the stack in correct order, and a deep access (access below the top of the hardware stack) is required to fetch expression inputs. The stack naturally acts as a cache, and strongly favors temporal locality, but without the drawbacks of cache evictions due to its logically infinite size. It stands to reason that most variables fit well into the model of temporally local caching, and most stack access will therefore be at or near the top of the stack. To gain insight into stack access behavior, we compile a number of benchmarks, execute them without a stack (using a directory of variables), and analyze the resulting traces using an oracle to keep track of stack access depth. We plot a histogram of stack access depths and relative frequency of stack access in Figure 3-2. The 80^{th} percentile is marked with a dashed line on each plot. As shown in the plot, most accesses indeed tend to be shallow, majority falling within the top 2 to 3 stack entries. The expected depth of access is quite low, ranging from 1 to 2 entries depending on the benchmark. More than 80% of accesses are shallower than 4 entries.

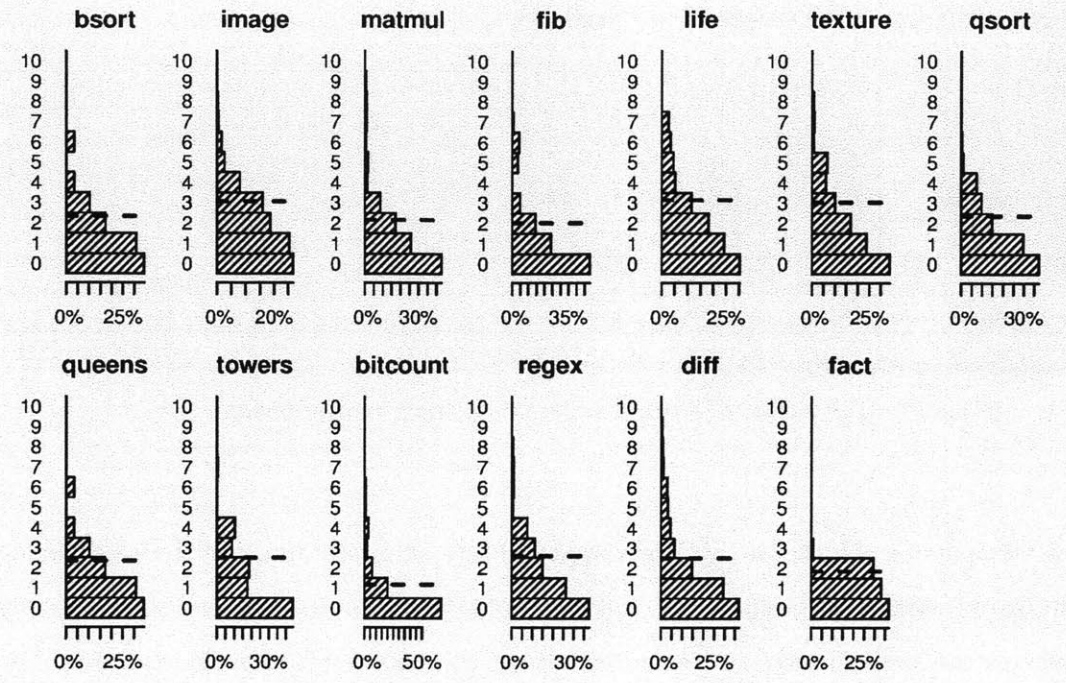


Figure 3-2: Histograms of stack access depths

3.3 Instructions for Stack Manipulation

The core ISA affords only a rather inefficient facility to perform deep accesses. To bring the i^{th} (i_n) element of the stack to the top, making it the 1^{st} entry, we can store the top n elements into memory somewhere, then load all except i_n in reverse order. We are now able to load i_n last, placing it on top without altering the order of the rest of the stack. Likewise, to swap elements i_n and i_m (assume without loss of generality that $m < n$), we are able to store n entries from the stack into memory, and read them back in reverse of the desired order. Clearly $2n$ memory accesses are necessary to perform an n-deep stack access using only the core ISA: a very expensive operation!

Recall from Section 3.2 that most accesses are quite shallow. Figure 3-2 shows the distribution of deep accesses with each benchmark. Most accesses tend to be shallow but in excess of 50% of accesses touch data below the top of the stack. It stands to reason that by introducing a facility to efficiently manipulate a region of a

EM^2 Instruction	Instruction Function in RTL
`pull` n	$F_{pull}(n) : S(i_0, i_1, \cdots, i_n, i_{n+1}, \cdots) \rightarrow S(i_n, i_0, i_1, \cdots, i_{n+1}, \cdots)$
`pull_copy` n	$F_{pull\_copy}(n) : S(i_0, i_1, \cdots, i_n, \cdots) \rightarrow S(i_n, i_0, i_1, \cdots, i_n, \cdots)$
`tuck` n	$F_{tuck}(n) : S(i_0, i_1, \cdots, i_n, i_{n+1}, \cdots) \rightarrow Stack(i_1, \cdots, i_n, i_0, i_{n+1}, \cdots)$
`tuck_copy` n	$F_{tuck\_copy}(n) : S(i_0, i_1, \cdots, i_n, \cdots) \rightarrow S(i_0, i_1, \cdots, i_n, i_0, \cdots)$
`drop` n	$F_{drop}(n) : S(i_0, i_1, \cdots, i_n, i_{n+1}, \cdots) \rightarrow S(i_0, i_1, \cdots, i_{n+1}, \cdots)$
`swap` n	$F_{swap}(n) : S(i_0, i_1, \cdots, i_n, i_{n+1}, \cdots) \rightarrow S(i_n, i_1, \cdots, i_0, i_{n+1}, \cdots)$
`main2aux`	$F_{main2aux} : \{S_{main}(i_0, \cdots), S_{aux}(\cdots)\} \rightarrow \{S_{main}(\cdots), S_{aux}(i_0, \cdots)\}$
`aux2main`	$F_{aux2main} : \{S_{main}(\cdots), S_{aux}(i_0, \cdots)\} \rightarrow \{S_{main}(i_0, \cdots), S_{aux}(\cdots)\}$

Table 3.2: EM^2 instructions for stack manipulation

few elements at the top of the hardware stack, we eliminate much of the inefficiency associated with stack accesses in common workloads. The histograms show that by allowing stack manipulation instructions to access values up to 4 elements into the hardware stack, we reduce most accesses to single-instruction operations. In fact, more than 80% of accesses are trivialized in this manner in all surveyed benchmarks. Table 3.2 enumerates the instructions added to the EM^2 ISA to explicitly manipulate the stack. The same instructions are plotted on a bullseye diagram in Figure 3-3.

There are, however, occasional variables located deeper than 4 elements on the hardware stack. Although optimizations introduced later in this thesis help mitigate this problem, it does not disappear entirely, so we must implement a method for deep access. Figure 3-4 shows that several classes of variables are particularly associated with deep accesses. Constants are frequently a source of overhead, as they are mapped to a single expression of the same value, and therefore must be communicated throughout the program much like frequently-used read-only variables. The problem of scheduling constant values is completely alleviated by a constant replication optimization detailed in Chapter 6. The other variable groups frequently associated with deep accesses are return addresses, loop invariants, and return values. These variables have low temporal locality and behave unlike intermediate values, which make up the majority or variables.

Instead of using memory, as shown in Section 3.3, we borrow from many exist-

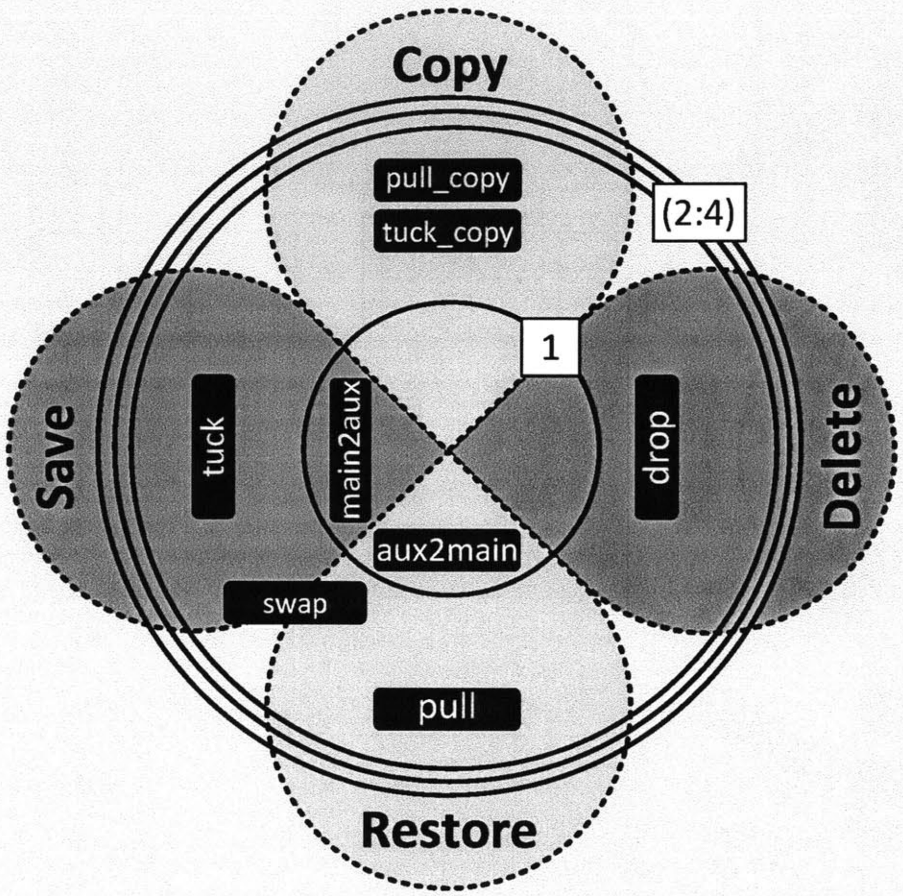


Figure 3-3: Bullseye diagram of EM^2 stack instructions

ing architectures [48, 14] and implement an auxiliary hardware stack with limited capabilities. Two additional instructions (`main2aux`, and `aux2main`) are introduced to move a value between the main stack and the auxiliary stack, as previously shown in Table 3.2. Not only does this auxiliary stack naturally accommodate many of the values that do not naturally gravitate towards the top of the stack when needed (return addresses, loop variables, etc.), but it also allows us to efficiently re-organize the stack without involving memory, as shown in Algorithm 1.

Although the algorithm looks expensive due to its abundance of control flow, it

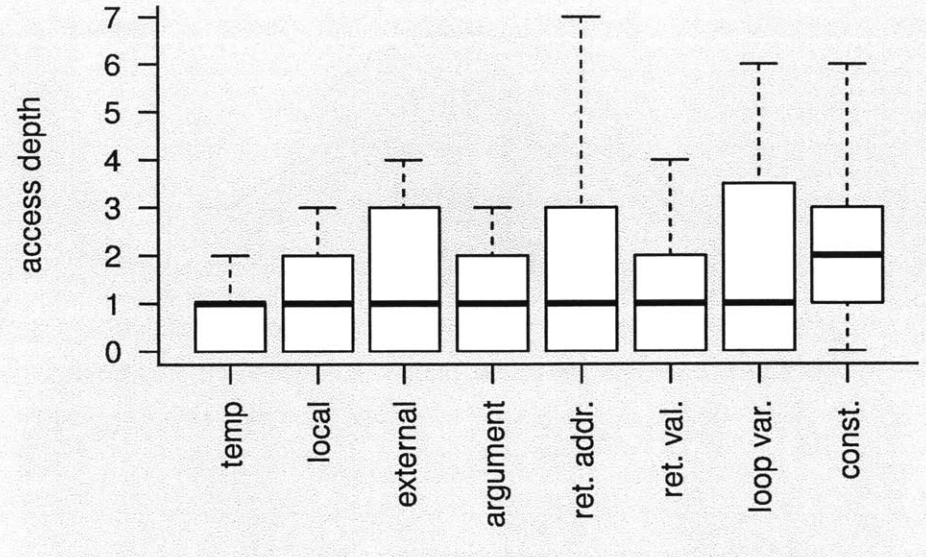


Figure 3-4: Depth of access for common variable types

is important to realize that all decisions are elaborated statically at compile time, and only stack manipulation operators are emitted (shown as emphasized). Similar algorithms for deep deletion, deep insertion, deep removal, and deep copy are trivially derived by modifying the deep swap algorithm accordingly. Figure 3-5 illustrates the overhead, in number of EM^2 stack instructions executed for each type of deep access. These operations are emitted in self-contained instruction sequences, and are therefore independent of other work in the system (cache interference is not considered), so such analysis without experimental data is appropriate. Deep access is trivial for depths up to 4 elements (by design, as discussed in Section 3.3), but the cost begins to increase linearly for greater depths. Copy operations are the most expensive. Tuck and pull are among the cheapest operators, while drop is the least expensive of the lot.

```
Input: $i$, $j \in N$ (assume without loss of generality that $i > j$);
Stack of the form $(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_{j-1}, \alpha_j, \alpha_{j+1}, \cdots, \alpha_{i-1}, \alpha_i, \cdots)$;
Output: Stack of the form $(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_{j-1}, \alpha_i, \alpha_{j+1}, \cdots, \alpha_{i-1}, \alpha_j, \cdots)$;
begin
    Let $n = 0$, $m = 0$;
    while $i_j$ is not topmost, and $i_i$ is $\geq 4$ elements deep do
        move a value from main to auxiliary stack;
        $n = n + 1$;
    while $i_i$ is $\geq 4$ elements deep do
        Let $d = min(4, \text{depth of}(i_i) - 3)$;
        tuck $i_j$ d elements down the stack;
        for 1 to $d$ do
            move a value from main to auxiliary stack;
            $m = m + 1$;
    // Aux. stack contains $n + m$ elements in reverse of their original order.
    swap $i_i$ and $i_j$;
    while $m > 0$ do
        Let $d = min(m, 4)$;
        for 1 to $d$ do
            move a value from auxiliary to main stack;
            $m = m - 1$;
        pull $i_i$ to the top of the main stack;
    // $n$ items remain on the aux. stack in reverse of their original stack order.
    while $n > 0$ do
        move a value from auxiliary to main stack;
        $n = n - 1$;
```

Algorithm 1: Deep-swap two stack entries using a secondary stack

3.4 Allocation and Scheduling of Local Expression Variables

So far, we are able to compile a C program into a sequence of expressions, as shown in Chapter 2, and have the vocabulary and facilities to allocate and rearrange variables on the hardware stack. We must now devise a scheme to judiciously insert stack manipulation instructions into the sequence of expressions to ensure the program executes correctly. In other words, we must schedule the abstract notion of "variables" used by the expression sequence onto concrete hardware resources, namely the hard-

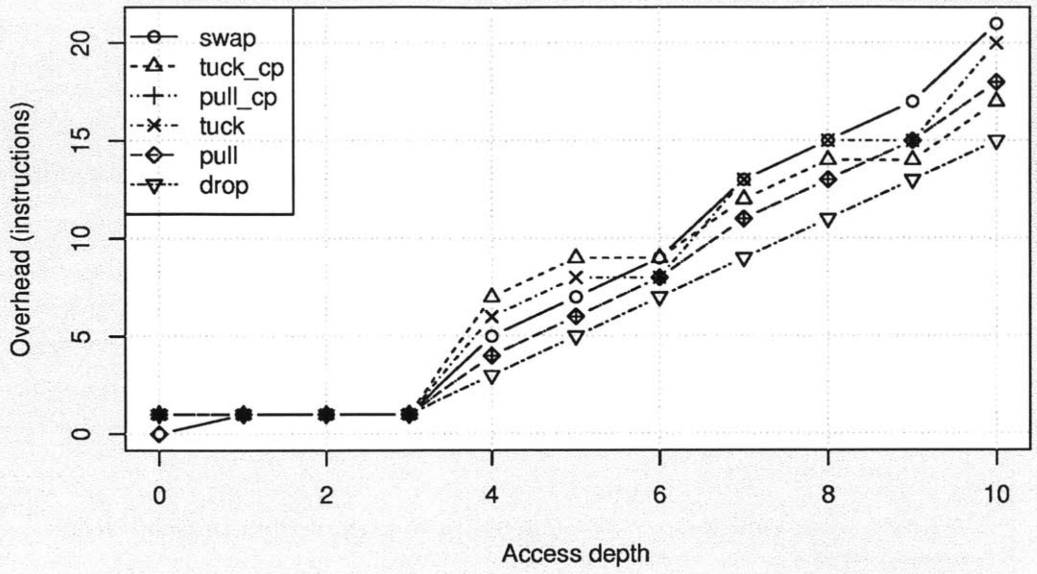


Figure 3-5: Analytical model for overhead of a deep stack access

ware stack. Normally, variables are mapped to a software stack frame and explicitly cached in a register file, a relatively easy problem given that software stack frame accesses allow arbitrarily deep access without an increase in access cost. An arbitrary scheduling of variables onto the hardware stack, however, often leads to an abundance of deep stack manipulation, which contributes to much unnecessary overhead.

In order to produce a clean, well-argued solution, we must avoid whole-program optimization and bound our discussion to a concrete subset of the program. We choose to localize the scheduling problem to "straight-line" code segments, otherwise known as basic blocks [39]. Doing so is advantageous because the absence of control flow ambiguity allows us to solve the variable allocation and scheduling problem in each basic block in isolation, since any basic block always executes in its entirety. Given any expression E in the program, we can determine the basic block it belongs to by considering the longest expression sequence B, such that $E \in B$, and B has no control flow ambiguity. It must be true that B is always evaluated as a unit. Control may only be transferred to the first expression in B; conversely, the last expression in B must transfer control to another basic block in the program. It must follow that a

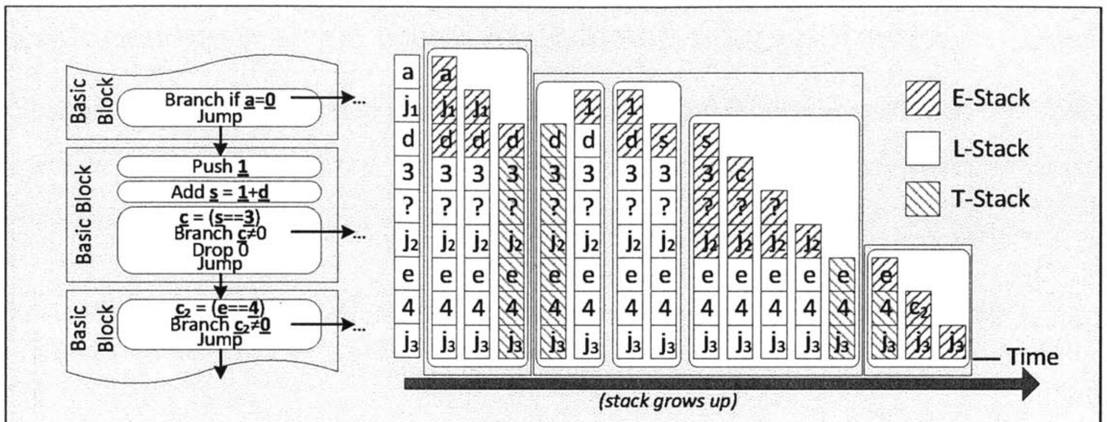


Figure 3-6: Variables in E-, L-, and T-stack regions

basic block is any sequence of instructions immediately following a terminal (`branch`, `jump`, or another control flow operator), and ending with the next terminal. We will discuss variable scheduling across basic blocks in Section 3.6.

3.4.1 Stack regions: E-stack, L-stack, and T-stack

Recall that we have previously defined a region on top of the stack used within an expression: the E-stack. The E-stack is scheduled implicitly by construction, as the instruction sequence comprising an expression is written statically by the compiler designer. We now define two additional stack regions: the local stack (L-stack), which is used to communicate variables within a basic block, and the transfer stack (T-stack), which is used to pass variables between basic blocks. Clearly, since the E-stack, L-stack, and T-stack are used to communicate variables among different organization levels in the program hierarchy, it must follow that these regions must not overlap. Logically, values may be moved between the stack regions, usually without any explicit instructions (simply by moving the imaginary boundary between the regions). Much like the stack frame, these regions reside on top of the logically infinite stack structure (which grows and shrinks with the call hierarchy), and together comprise the software stack frame for a given function (Shown in Figure 3-6). In this section, we will limit our discussion to L-stack scheduling, deferring T-stack scheduling until Section 3.6.

3.4.2 Methodology for Local Expression Variable Scheduling

Given the definition of the L-stack region, we are able to better formulate and expand the invariant defined in Section 3.1:

- The E-stack must be empty.
- The L-stack immediately prior to execution of an operator $F(i_1 \cdots, i_2 \cdots) \rightarrow \{o_1, o_2, \cdots\}$ must be of the form $(i_1, i_2, \cdots)$.
- The L-stack at any point while executing a basic block must be precisely the set of all live variables [4] at this point.
- The T stack must be empty.

These invariant statements must be maintained in order to correctly execute a sequence of expressions comprising a basic block. This section will survey several approaches: a naive greedy in-order approach, an optimal global solver, and an approach based on a well-known stack scheduling algorithm.

3.4.3 In-Order Stack Allocation

The most straightforward approach to variable scheduling is to traverse the sequence of expressions in order, and manipulate the L-stack into a correct ordering for the next expression to execute. To this, we explicitly pull the expression inputs to the top of the stack before each expression. We must be careful to maintain the set of live variables [4] before and after each expression to avoid deleting a value prematurely, or leaving unused data on the stack when the basic block terminates. The algorithm used in this approach is shown in Algorithm 2.

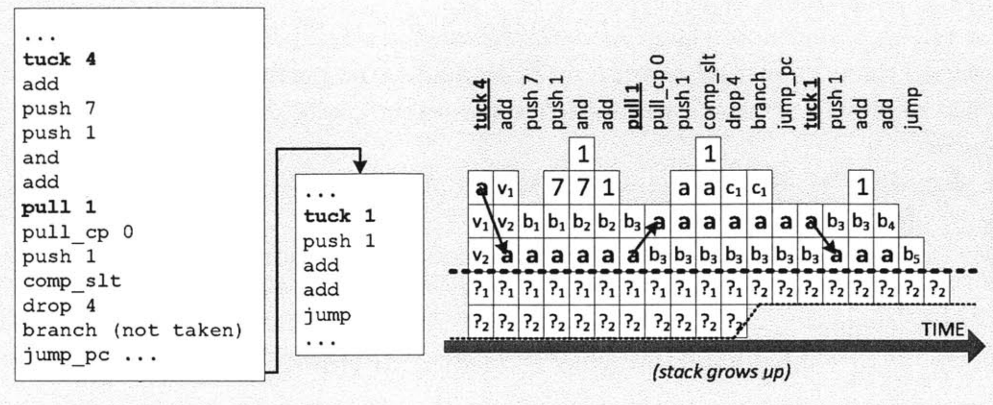
This algorithm traverses the set of expressions once, and also requires a live variable analysis [4] (a single separate pass over the set of expressions, accumulating live variable list at each expression), so overall algorithm complexity is linear in time.

```
Input: Basic block $B = (F_1, F_2, \cdots)$;
Let $(i_1, i_2, \cdots)$ denote the prior variables on the stack before $B$ executes;
Assume that the last expression in $B$ populates the posterior T-stack, and
takes as input $\cup$(T-stack variables $\forall$ control flow paths originating from $B$);
begin
    for $\forall F_i : (i_1, i_2, \cdots) \rightarrow (o_1, o_2, \cdots) \in B$. do
        for $\forall i_i$ of $B$ in reverse order: do
            if $i_i \in L_i$, where $L_i$ is the live set after $F_i$ then
                immediately prior to $F_i$, copy $i_i$ to the top of the stack;
            else
                immediately prior to $F_i$, pull $i_i$ to the top of the stack;
        for $\forall o_i$ of $B$: do
            if $o_i \notin L_i$, where $L_i$ is the live set after $F_i$ then
                immediately after $F_i$, drop $o_i$ from the stack;
```
Algorithm 2: In-order L-stack scheduling

The performance of the algorithm, however, is a more complex issue. Clearly, the algorithm makes no effort to reduce scheduling overhead, blindly rearranging the stack as needed at every expression. Instead of reasoning about asymptotic bounds for worst-case behavior of the scheduling logic, we directly measure the overhead with a set of representative benchmarks in Section 3.4.6.

3.4.4 Optimal Stack Allocation

To better understand best-case overhead of the local variable scheduling for the EM^2 architecture, we also consider a scheduling algorithm built to find a variable assignment with lowest overall cost. Given a basic block $B = (F_1, F_2, \cdots)$, where each expression is of the form $F_i : (i_1, i_2, \cdots) \rightarrow (o_1, o_2, \cdots)$, we observe that a variable placed onto the stack maintains its position, even though the stack may grow or shrink above it (the stack remains unchanged below, except in cases of deep accesses). This is illustrated in Figure 3-7. By artificially treating a slice through the stack as horizontal (thick dotted line in the diagram, the "horizon"), we may better visualize the movement of values in the stack adjacent to the imaginary horizon. Overhead in the form of stack operations is only introduced if the variable is "moved" via `pull`,



```
...
tuck 4
add
push 7
push 1
and
add
pull 1
pull_cp 0
push 1
comp_slt
drop 4
branch (not taken)
jump_pc ...
```

```
...
tuck 1
push 1
add
add
jump
...
```

Figure 3-7: Stack behavior: deep variables maintain their position

tuck, or other stack instructions. Some movements are more expensive than others, as dictated by the depth of access; we previously presented the costs associated with deep access in Section 3.3. Given a set of start and end positions (the definitions and uses), we find a set of "moves" such that the total cost of these moves is minimized. Each variable must remain below the top of the stack during its lifespan.

Normally, it is impractical to exhaustively search the space of schedules, so a compiler would employ various heuristics to find an acceptable, but not necessarily optimal, schedule [53]. When solving for the optimal schedule, we cannot resort to these tricks, and instead use a number of search optimizations to prune the space of optimizations. We employ a subset of the methods used in previous work [38] to dramatically trim the search space, allowing the optimizer to run to completion in minutes for most benchmarks. We use *branch and bound* to quickly discard "bad" schedules if the cost of a partial schedule exceeds the minimal cost observed for a full schedule. We also eliminate tree-shaped subgraphs, from the scheduling problem, as these are trivially reduced by enumerating their operators in reverse polish order. In the few cases where optimal scheduling took in excess of several minutes to schedule a basic block, we manually schedule a portion of the basic block and re-run the scheduler.

3.4.5 Modified Koopman Stack Allocation

Koopman's stack scheduling algorithm [32, 31] is an early attempt to solve the scheduling problem for a stack architecture. Koopman does not consider the stack scheduling problem at a basic block level, or any well-defined subset of the program ("I just used ad-hoc techniques as necessary" [32]). He post-processed the textual output of GCC and inserted stack operations in place of loads and stores, where possible, to "promote" variables to the stack, and out of memory. The algorithm works by annotating all instructions as producers and consumers of variables by enumerating the variables defined and used in the code sequence. Koopman then creates a sorted queue of variables, ordered by the distance between the definition and use. In queue order, he tucks the variable to the bottom of the stack at its point of definition, and pulls it to the top at the point of its use.

In this thesis, we adapt Koopman's algorithm for our L-stack scheduler. We modify the algorithm to add efficiency by observing that a variable need not be copied to the bottom of the stack, but only placed below any other currently scheduled variables. We define the set of all variables scheduled by the algorithm in a basic block b to be exactly the L-stack region for b. We also extend the algorithm by introducing the notion of deep uses. Consider an addition operator: it consumes two variables: a left-hand side (LHS) and right-hand side (RHS). While LHS is read from the top of the stack, RHS implies a 1-deep access. To correctly handle these operators, we always give preference to def-use pairs containing a RHS use. Extending this to expressions with multiple inputs, we always give preference to variables with the deepest uses when performing local scheduling. Algorithm 3 details the modified Koopman's stack scheduler, as used in this work. The algorithm is linear, give a sorted collection of def-use pairs, giving a total complexity of $\mathcal{O}(NlogN)$ in time.

```
Input: Basic block $B = (F_1, F_2, \cdots)$,
where $F : (u_1, u_2, \cdots) \rightarrow (d_1, d_2, \cdots)\ \forall F \in B$;
1). enumerate all variable defs and uses;
2). $P_d = \{$def of $v$, subsequent use of $v\}\forall v$;
3). $P_u = \{$use of $v$, subsequent use of $v\}\forall v$;
4). $Q = P_d \cup P_u$, ordered by span − RHS depth;
Let $S$ be an array of empty stacks, such that $|S| = |B|$;
begin
    for $p = (v_d, v_u) \in Q$ do
        Let $v$ denote the variable;
        Let $id_d$, $id_u$ denote the indexes of the expressions spanned by the pair;
        $d_d = |S_{id_d}|$;
        $d_u = |S_{id_u}|$;
        tuck $d_d$ $v_d$ to the top of the L-stack.
        for $i = id_d$ to $id_u$ do
            Append $v$ to $S_i$.
        if $v_d$ is a use then
            (pull-copy $d_u$), moving $v_d$ to the top of the L-stack, if needed.
        else
            (pull $d_u$) moving $v_d$ to the top of the L-stack, if needed.
```
Algorithm 3: Modified Koopman's scheduler

3.4.6 Evaluation of Local Stack Allocation Algorithms

To evaluate the three schemes for scheduling expression variables onto the hardware stack, we compile a diverse set of benchmarks and schedule all basic blocks in each program using each of the three schemes proposed in this section. When executing the benchmarks, we must use a dictionary of variables to correctly communicate values across basic block boundaries (this sub-problem will be addressed later in this chapter, in Section 3.6). Analyzing the traces, we measure overhead as the ratio of all instructions, including stack manipulation, to the instruction count in the sequence of expressions only. This quite literally gives the overhead of scheduling in each of the surveyed benchmarks. The results are shown in Figure 3-8. The modified Koopman scheme performs nearly as well as the optimal scheduler, but runs in $O(NlogN)$ time, where N is the number of variables in the basic block. The optimal scheduler, although marginally more efficient in general, takes exponential time, and is clearly not practical for a compiler application. The greedy in-order scheme performs poorly

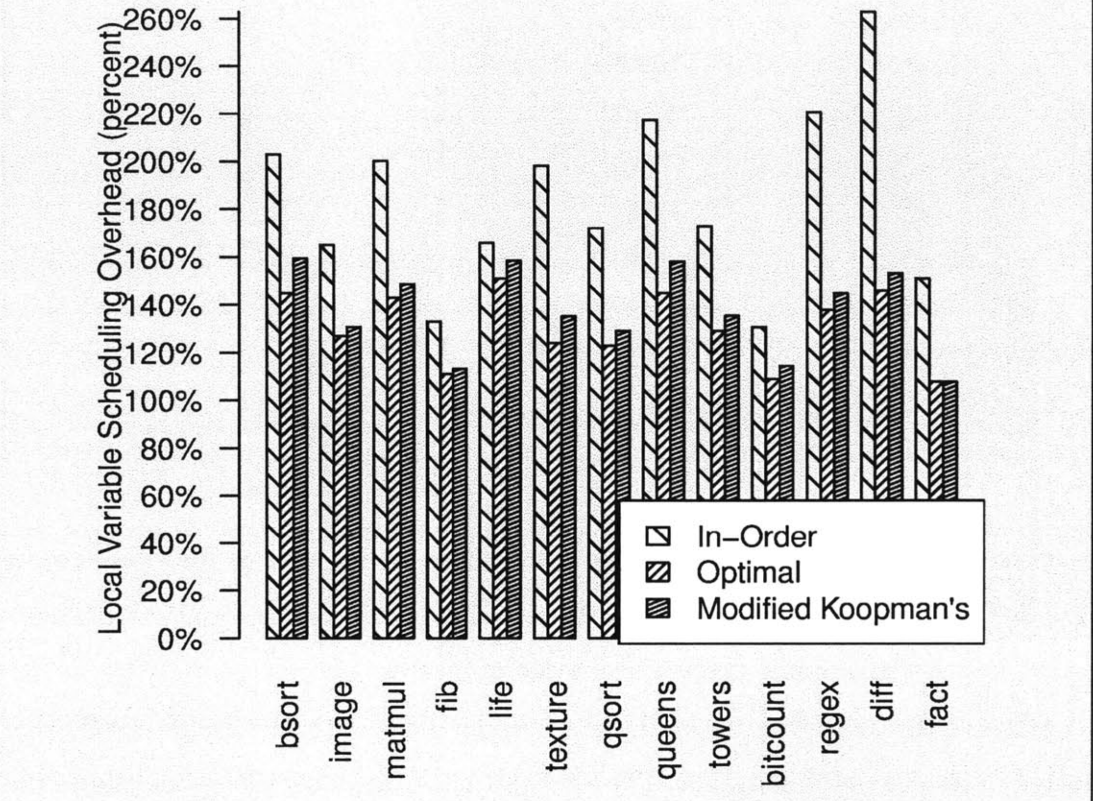


Figure 3-8: Overhead associated with surveyed L-stack scheduling schemes

in comparison. The modified Koopman scheduler is clearly the best choice among algorithms surveyed for the EM^2 compiler, and adds on average 0.6 instructions per expression (within 11% of optimum) to schedule all local variables.

3.5 Expression Reordering for Efficient Scheduling

The portions of the compiler backend implemented so far give us a legal sequence of expressions for each basic block, but the ordering within the sequence is dictated by the front-end, and may not be optimal for mapping to a hardware stack-based architecture. Consider for example a sequence of expressions reducing several values to a sum, as shown in Figure 3-9. For the purposes of this discussion, we overlook any algebraic optimizations that may trivialize the example. Clearly the order of

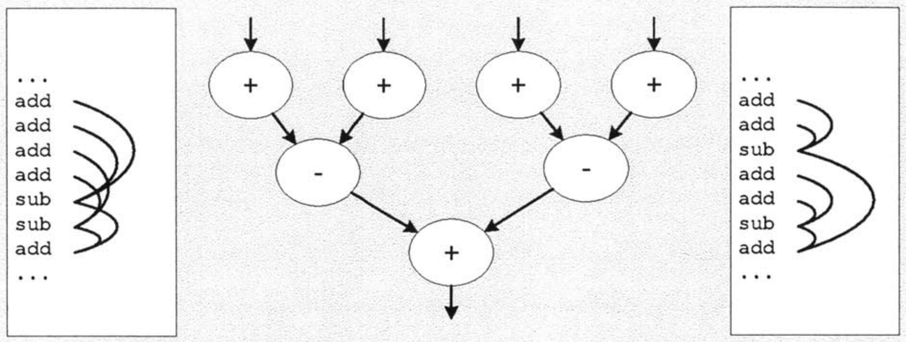


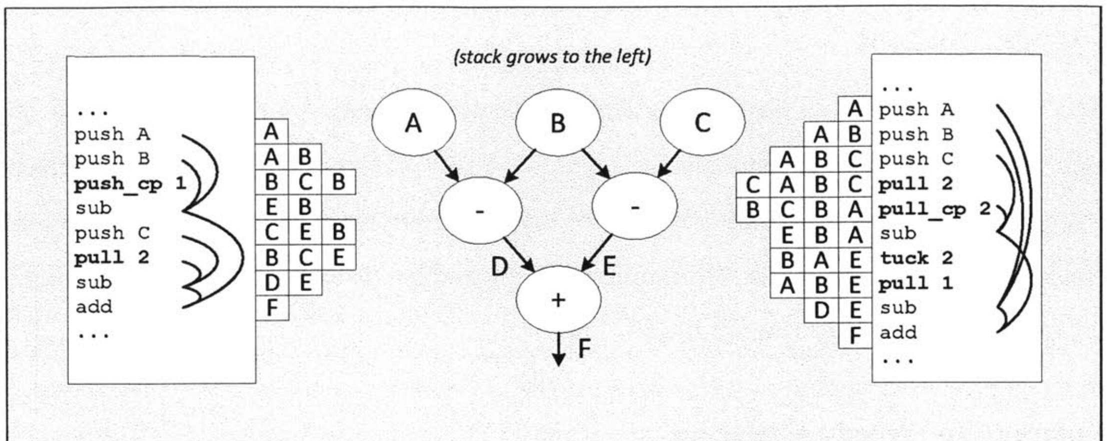
Figure 3-9: Operator order is flexible, up to dependencies

operations is largely irrelevant as long as dependencies are met in that any correct ordering produces the same result, but the scheduling overhead associated with some orderings may be an improvement over other sequences.

Observe that a perfect stack schedule is trivial for a data flow graph (DFG) presented in reverse polish notation. The left hand side of any operator is mapped to the stack above the right hand side, so by expanding the right hand side of the root operator completely before expanding any of the left hand side, we ensure that no stack manipulations at all are necessary to correctly execute the set of expressions. Unfortunately, real-world workloads seldom lend themselves to perfect schedules: many variables are used more than once, and often at very different levels in the DFG. These "imperfections" make an ideal ordering (an ordering requiring no stack manipulation at all) unlikely, although very low-overhead orderings are achievable.

3.5.1 Topological Sort to Enumerate Expression Sequences

Variables are simply logical entities that enumerate dependencies. In other words, variables do not correspond to explicitly managed, long-lived state. Instead, they simply enumerate explicit dependencies in the data flow graph: an expression defining some variable v must execute before another expression that uses v. If a variable is used by multiple expressions, these may be executed in any order provided their



```
...
push A
push B
push_cp 1
sub
push C
pull 2
sub
add
...
```

```
...
push A
push B
push C
pull 2
pull_cp 2
sub
tuck 2
pull 1
sub
add
...
```

Figure 3-10: Data flow graph with efficient and inefficient linearization

dependencies are met. In fact, any correct sequence of expressions is a linearization (topological sort) of the DFG induced by its variables. It is important to note that there exist dependencies in addition to those made explicit by variables: implicit dependencies occur due to data being passed via memory, or via procedure calls. Although it is possible to analyze and handle dependencies with sufficient effort, this work takes a conservative approach (thereby avoiding a complex, open research problem) and maintains memory access interleaving, as well as the order of function calls relative to each other and to memory accesses. Using this knowledge, we can easily enumerate the entire space of correct orderings by enumerating topological sorts of the DFG, taking care to reject any sequence that violates implicit dependencies. Figure 3-10 shows a DFG with an efficient linearization, and an equivalent but severely inefficient linearization. This section surveys a number of different approaches to selecting an efficient ordering and discusses the complexity and other characteristics of each. Finally we evaluate these approaches against a set of representative benchmarks in Section 3.5.2. As a baseline, we evaluate the unmodified expression sequence as emitted by the compiler front-end. This approach has no overhead in terms of compile time, but has unknown quality of result, as the front-end compiler does not optimize for minimal stack scheduling overhead.

Greedy Ordering

The simplest scheme constructs a topological sort in order, one expression at a time. Each time, the expression $F_i : (i_1, i_2, \cdots) \rightarrow (o_1, o_2, \cdots)$ is selected such that $(i_1, i_2, \cdots)$ minimizes the edit distance for current order of variables on the hardware stack. By looking ahead and maintaining a live variable set, we avoid all backtracking, meaning the algorithm runs in linear time.

Bottom-Up Greedy Ordering

The reverse of the previous scheme builds the topological sort from the final expression backward, observing that the last expression of a basic block is a terminal (transfers control flow), and is therefore fixed. The cost metric is the same as in the forward greedy ordering approach. This approach dramatically reduces the search space, and therefore the amount of backtracking, and yields better results: most basic blocks implement a reduction, whereby a set of variables is combined by various expressions into a smaller set. By scheduling in reverse order, the greedy scheduler is better able to explore the graph depth-first, as if it were a tree, thereby finding generally better schedules.

Optimal Ordering

The optimal scheme enumerates all topological sorts, schedules them all using the modified Koopman algorithm, and selects the lowest cost sequence of expressions (using an edge intersection heuristic to eliminate obviously bad linearizations early). This approach is extremely costly in both time and memory, and is not fit for implementation as part of a compiler. It is included to show an upper bound on the quality of scheduling results.

Optimizing Bottom-Up Ordering

The optimizing reverse ordering scheme is a hybrid between the reverse greedy scheme and the optimal ordering. Much like the optimal ordering, it enumerates the space induced by topological sorting over the DFG, but unlike the optimal scheme, it is not allowed to explore the entire space. Instead, optimized reverse ordering proceeds breadth-first, selecting only branches within a parameterized cost margin of the cost of a greedy choice. Overall cost of a sequence is evaluated by scheduling it using the modified Koopman algorithm, and calculating the overhead. We benchmark several variants of the optimizing bottom-up scheme by varying the margin parameter. The margins evaluated are: $\{2, max(3, 10\%), max(4, 20\%), max(5, 30\%)\}$. In the case of margin$=$ 2, we simply select 2 lowest-cost branches greedily at each decision point when building the set of linearizations. For all other margins, we use a percentage of the lowest cost, but also take care to limit the absolute number of branches taken (to avoid long run times in case of reductions where all decisions carry the same overhead). The complexity of this scheme varies with how much of the search space is actually evaluated, dictated by the choice of margin. This scheme is bounded search in the entire space of possible linearizations, ordered by a greedy local heuristic.

3.5.2 Evaluation of Expression Reordering Schemes

To evaluate the expression ordering schemes, we compile a diverse set of benchmarks, reorder the expressions in each basic block using each of the schemes described in Section 3.5.1, and schedule them using the modified Koopman algorithm. When executing the benchmarks, we must use a dictionary of variables to correctly communicate variables across basic block boundaries, as this sub-problem will be addressed later in this chapter in Section 3.6. Analyzing the traces, we define overhead as the ratio of all instructions (including stack manipulation) to the instruction count in the sequence of expressions only. The results are shown in Figure 3-11. Surprisingly, the unmodified expression sequence emitted by the front-end yields relatively good schedules, sometimes outperforming the reverse greedy approach. The forward greedy approach performs poorly across the board. In many cases, the optimizing reverse ordering scheme approaches the efficiency of optimal ordering, even with a small margin. We chose to implement the latter, with margin$=max(3, 10)$ as part of the standard compiler back-end flow, achieving good, low-overhead linearizations

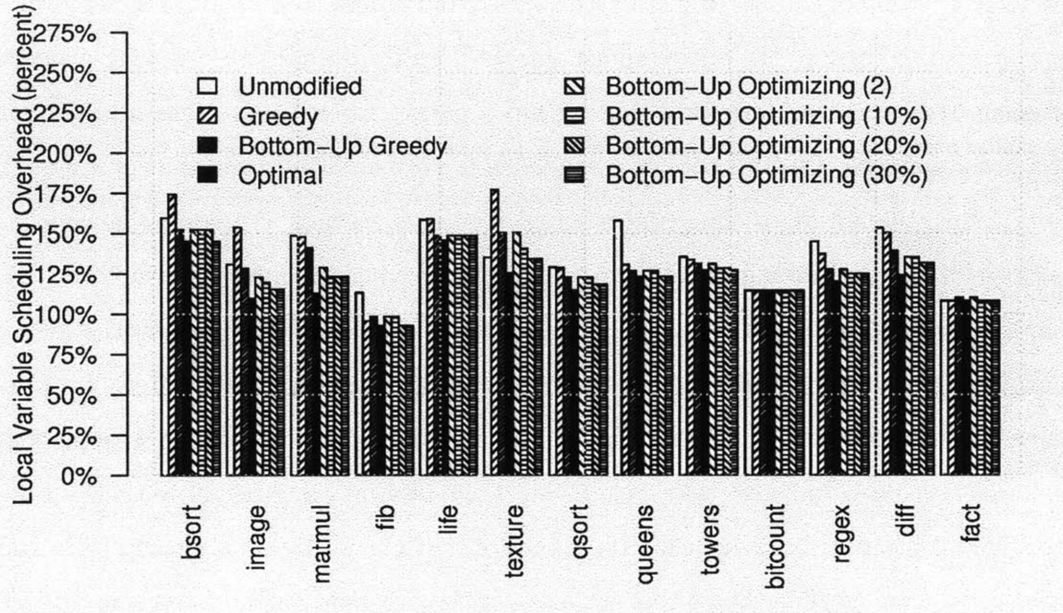


Figure 3-11: Local scheduling overhead with surveyed expression re-ordering schemes

quickly for most benchmarks.

It is important to note that the order of the expression sequence in a basic block has far-reaching implications in the Execution Migration Machine, beyond efficiency of stack scheduling, because it impacts migration behavior and potentially causing or alleviating a high eviction rate. Various optimizations can be applied by altering the cost function of the optimizing reverse ordering approach, as explored in Chapter 6.

3.6 Variable Scheduling Across Basic Blocks

The last sub-problem we need to solve in the EM^2 compiler backend is scheduling of T-stack variables. Although we are able to produce high-quality schedules at a basic block level, we must pass variables between basic blocks to correctly execute a program. Much like in the case of basic block scheduling, we must maintain an invariant, whereby before we transfer control to a basic block B, the T-stack must consist of all variables used by posterior basic blocks of V, but not defined by them:

Before a basic block b is evaluated, the T-stack must contain precisely $\{def(B_{prior} \cap (use(b) \cup (use(B_{posterior}\}$, where:

- $prior(b) = \{b_{prior} | b_{prior} \in B$, and $b_{prior} \rightarrow b\}$, meaning there exists a control flow path (consisting of one or more edge) that takes b_{prior} to b.
- $posterior(b) = \{b_{posterior} | b_{posterior} \in B$, and $b \rightarrow b_{posterior}\}$, meaning there exists a control flow path by which $b_{posterior}$ is reachable from b.
- $use(b)$ and $def(b)$ are sets of variables used within b and defined within b, respectively.

Two approaches are possible: either the T-stack contains exactly the set of variables stated in the invariant, and the source basic block must populate the T-stack differently depending on the control flow decision, or the T-stack contains the union of the T-stack required sets for all control flow targets of b. In case of the latter, each basic block must drop elements depending on the source of the basic block. We implemented the former scheme, as the control flow expression is already well-equipped with the required branch structure and information to correctly populate the T-stack with relevant variables.

A more complex question is the order of variables in the T-stack. Here, an optimal schedule is difficult to formulate, as a given T-stack schedule affects overhead in not one but several basic blocks - a very complex optimization problem to address, and not one we can evaluate statically due to overheads being highly dependent on the control flow paths taken by a program.

To produce a good T-stack assignment, we schedule basic blocks in reverse: the CFG of a function has well-defined return points (sinks on the graph), which execute last regardless of the path a program takes through the CFG. When ordering basic block expressions and applying the modified Koopman algorithm to schedule the L-stack (as detailed earlier in this chapter), we are able to modify the terminal expression to correctly populate the T-stack. This is easy because posterior basic blocks will have been scheduled, making the live variable analysis trivial. In case of loops, a basic block must be scheduled with incomplete knowledge of the output T-stack (because it has a CFG edge to itself). We optimize such basic blocks to loop on themselves - a sensible approach given the fact that most loops run for many iterations. Using this heuristic, we are able to order the T-stack variables in a way that incurs least overhead in the target basic block, and achieves good results overall. It must be noted, however, that this approach is merely a heuristic, as whole-program optimization at this level is expensive and highly complex.

We do not evaluate the T-stack scheduler in isolation because the T-stack performs local scheduling as part of its algorithm. We therefore evaluate the entire compiler backend by compiling a set of representative benchmarks, order and schedule the T-stacks as detailed above, using optimizing reverse ordering and the modified Koopman algorithm to perform local scheduling. We profile the benchmarks using an ISA simulator for the EM^2 project. We compute total overhead by tallying the ratio of all instructions executed relative to non-stack instructions, as before. The results are shown in Figure 3-12. The overall scheduling overhead is approximately 40% for the majority of benchmarks, with the notable exception of `life`, which consists of predominantly short basic blocks and a lot of control logic, causing additional overhead in each phase of variable scheduling due to high control flow ambiguity. `matmul`, `regex`, and `diff` were compiled from large, convoluted bodies of source code, and also exhibit high levels of control flow ambiguity. `texture` has long, arithmetic-heavy basic blocks, and is scheduled relatively efficiently. On average, 5.2 stack manipulation instructions are needed per basic block. The scheduling overhead added by the T-stack pass is about 12% on average (T-stack scheduling is necessary to make a program executable). It is obvious that scheduling at the basic block level is more efficient per variable than inter-block scheduling. This makes sense because a basic block can be scheduled in isolation, while T-stack scheduling choices can have far-reaching implications, and are constrained by control flow ambiguity. If done in the context of just-in-time compilation, the scheduler may be able to process entire traces, achieving high-quality schedules for frequently taken paths through the control flow graph, improving overall scheduling efficiency.

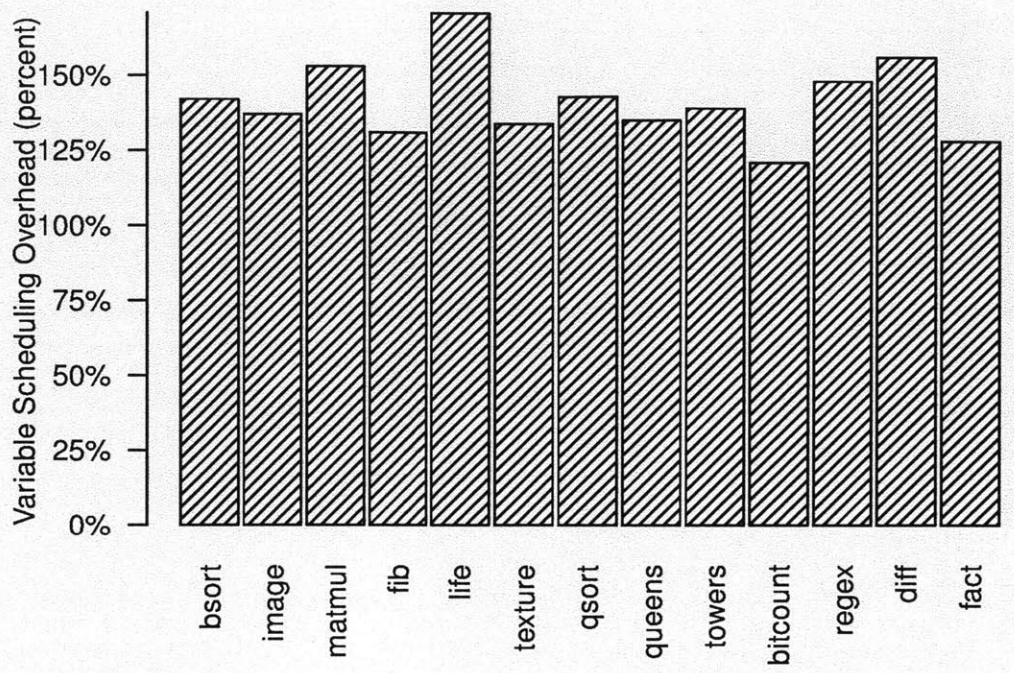


Figure 3-12: Total overhead due to stack scheduling in compiled programs

3.7 Summary

In this chapter, we explored the set of mechanisms by which symbolic state (variables) is scheduled onto hardware resources. More specifically, we discussed expression ordering and scheduling of variables within and across basic blocks. We defined the notions of expression, local, and transfer stack regions, and used these to precisely formulate invariants and optimization goals throughout this thesis. We evaluated a diverse set of algorithms for each phase of the compiler backend, and selected the modified Koopman algorithm for scheduling variables within each basic block, a flexible algorithm driven by a configurable cost function for expression ordering, and a simple set of heuristics for variable passing between basic blocks. The compiler backend is now able to implement C programs for the Execution Migration machine (symbolic, as instruction encoding has not yet been discussed), achieving reasonably low overhead incurred due to variable scheduling onto the EM^2 architected state.

Chapter 4

EM^2 Instruction Set Encoding for a Dense Instruction Stream

So far we discussed an entire compiler backend which, in collaboration with a stock front-end and platform-independent optimizer, translates programs written in C to the EM^2 instruction set architecture. While this system is enough to simulate the execution of programs, and to conduct many architectural studies, we cannot yet build a binary executable: we have not designed the ISA encoding to be interpreted by the EM^2 hardware. Our choice of encoding is very important, as it affects the performance of a system in two ways: decoder complexity directly affects the rate at which instructions can be issued, and encoding density affects code size, and therefore instruction cache efficiency and DRAM bandwidth utilization, thereby affecting performance of the entire system. Figure 4-1 shows the role of instruction encoding in the lager context of a complete compiler backend. A sparsely encoded ISA results in excessive code size, which results in low instruction cache efficiency, and excessive bandwidth required by the network and memory sub-system. On the other hand, a complex encoding requires a complex hardware decoder, which may limit overall system performance due to latency, power, area, or other constraints. A good encoding is both easily decodable and dense, improving the instruction cache miss rate, bandwidth, and reducing the overhead incurred by additional instructions due to variable scheduling, as discussed in Chapter 3

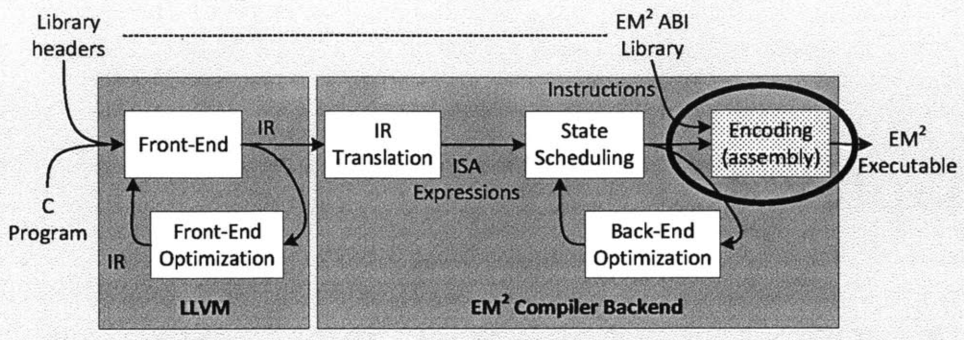


Figure 4-1: High-level role of instruction encoding in the EM^2 compiler

In this chapter we discuss the encoding problem in abstract terms and derive an upper bound on instruction stream density (lower bound on code size) with and without microcoded "super instructions". We then propose and evaluate a set of different encoding mechanisms, discuss the associated decoder complexity, and show that a naive but compressible encoding performs relatively well, while requiring no complex decoder logic. As a result, we can achieve a dense instruction stream, which helps mitigate the overhead of explicit scheduling of data. Finally, we show that the proposed encoding scheme reduces variable scheduling overhead on code size to only 12% relative to a naive encoding.

4.1 General Practices for Instruction Encoding

Reduced Instruction Set Computer architectures (RISC [44]), have similar (albeit register-based) instruction sets to the EM^2 ISA. These architectures typically encode their instructions as 32-bit binary sequences, but it is important to note that RISC instructions are relatively dense [28]. They encode not only the operator, but also the source and destination registers to perform the operation on, and often an immediate (instruction-encoded constant) as well.

A sparse encoding results in large code size, a particularly sensitive point for a stack-based architecture. Unlike RISC-type architectures, a program running on EM^2

incurs some code size overhead by needing to explicitly manipulate the hardware stack. The jury is still out on whether this overhead is significantly worse than register loading and spilling [7]. Although this may be due to a severely understudied problem of compiling for a hardware stack-based architecture, the colloquial wisdom is that stack architectures tend to execute more instructions to perform the same work as a register file architecture [44]. A dense instruction encoding is therefore very important to compensate for any code density lost due to stack management. A good instruction encoding also allows for a simple, high-performance decoder: for example, a context-free encoding allows for a stateless, combinational circuit, whereas any context awareness forces the decoder stage to consume a clock edge, and employ conservative added delays for hold time safety.

4.2 The EM^2 ISA is Sparse

The EM^2 instruction set is obviously sparse in comparison with a RISC-like instruction set: where each RISC instruction encodes a lot of information, most EM^2 instructions encode only the operator selection, although a few also encode one or two constants. In this section, we attempt to reason about the EM^2 instruction set density in abstract terms, deriving a lower bound for the number of bits required per instruction, on average. We do not assume constant instruction bit width - doing so would negate the highly sparse nature of most stack instructions.

4.2.1 Ideal Encoding as a Baseline

To establish an ideal, an upper bound on instruction density, we measure the average ideal instruction bit width in a set of benchmarks. The ideal encoding density varies with differences in the instruction stream. For example, an encoding may theoretically offer a 1-bit representation of the add instruction, but a 10-bit representation of subtraction, causing addition-rich traces to be more densely encoded than subtraction-rich ones. For this thought exercise, we ignore the decoder implementation, and assume the memory subsystem is capable of fetching instructions of any bit width and any alignment. To maintain relevance, however, we require the instruction set encoding to be context-free. We compile a representative set of benchmarks, execute them symbolically, and analyze the traces to determine the ISA information density.

Given a trace of EM^2 instructions, we need to determine the average bit width of each instruction if the trace is encoded to a minimal number of bits, with the constraints discussed above. In other words, we are given a set of instruction commands I and a trace detaining their relative frequency of the form $T = \{i : f\}$, such that $T(i) = f$, and $\sum T(i) \forall i = 1$.

With this information, we want to formulate a bijective mapping $F : i \to b$ from I to Z_N, where $N = |I|$ such that the cost function $\frac{1}{N} * \sum(T(i) * log_2(b)) \forall i$, where $b = F(i))$ is minimized. In English, we want to find the bijection (decodable encoding scheme) F that minimizes average bit-width of the instructions in a given trace T. This is equivalent to the space of permutations over Z_N, which has asymptotic complexity of $\mathcal{O}(N!)$, where N is the number of distinct instruction commands (for example two branches with different offsets are distinct instruction commands) so it is not practically enumerable for our purposes.

Fortunately, there is a very simple and effective heuristic: because we want to minimize a *sum*, we need only to minimize each of its terms. To do so, observe that $log_2(i)$ is monotonically increasing over $i \in Z_N$, so we want to assign the most frequently occurring instruction commands to the smallest numbers in Z_N. We can do this in constant time, given T is stored as a sorted histogram. Figure 4-2 shows a whisker plot of the instruction bit widths given a minimal encoding optimized to most compactly represent a given benchmark.

It is clear from this evidence that the stack ISA is very sparse: compared to a typical RISC instruction set [28, 54], which densely utilizes all 32 bits of each instruction word, the ideal encoding scheme only uses 6 to 8 bits per instruction on average, with maximum (encountered in one of the benchmarks) bit width being 12 bits. The bit width of the widest instruction in this encoding is 19 bits (instructions not present in the trace are encoded also!). It is also evident that our instruction

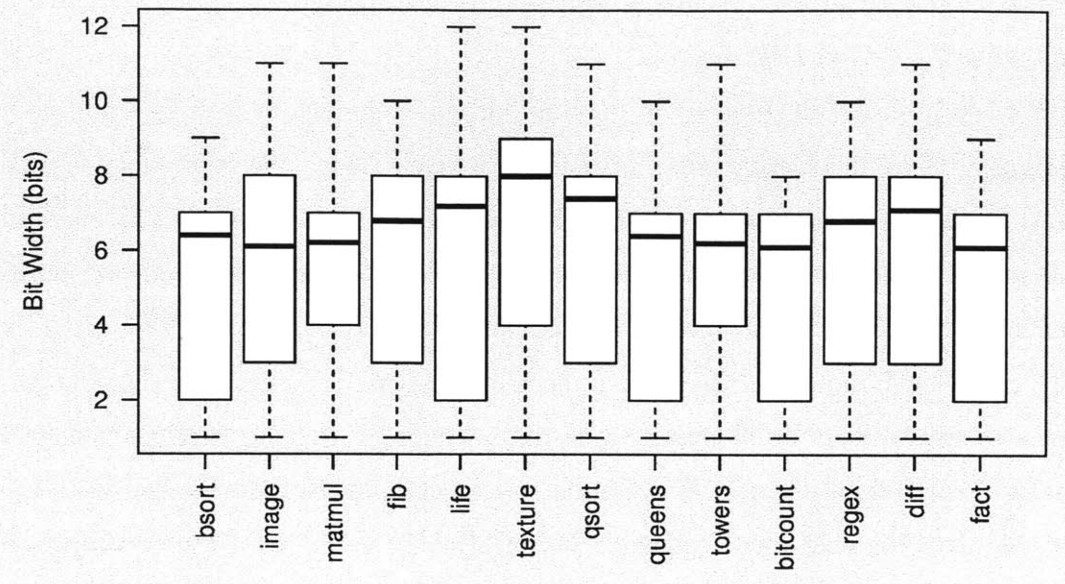


Figure 4-2: Distribution of instruction widths with ideal single-cycle encoding

set benefits from variable bit width encoding: although the average bit width of the instruction stream is relatively close to the minimum, some infrequently occurring instructions require significant additional encoded information, which manifests itself on the figure as a tall 3^{rd} quartile measurement.

Although very efficient, the ideal scheme is not practically implementable, as it is designed with no regard for decoder complexity. Moreover, the encoding is derived separately for each benchmark trace, and is not representative of all programs. The encoding optimized for the `texture` benchmark performs relatively poorly when applied to the `fib` trace, showing an encoding of 14 bits on average - a much less favorable statistic than the ideal encoding, which uses only 6 bits on average. The exact encoding reveals little interesting information, aside from an observation that small branch offsets and shift amounts are far more common than large ones.

4.2.2 Ideal Microcoded Encoding

Some instruction sets include "super instructions" - special binary sequences that trigger a sequence of instructions to be executed. This scheme allows sequences of instructions frequently occurring together to be encoded as a unit, potentially improving the overall code density.

We establish an ideal encoding baseline for this approach as well by building a dictionary of all instruction sub-sequences in a given trace. We must take care to limit the set of expressions we consider, however. If we were to encode sequences of unbounded length, any program would be reducible to a single half-used bit, dereferencing a microcoded sequence of instructions that implement the entire program (this is obviously not practical). For a more informative discussion, we consider microcoded sequences of finite length, up to and including $L \in N$ instructions, where L is a fixed variable. Given $L \in N$, we build an encoding, in the same manner as in the previous exercise, encoding a sequence whenever doing so improved the cost function (the trade-off here is that by encoding extra "super instructions", the cardinality of the instruction command set increases, increasing the maximum instruction bit width, thereby affecting the average). Whenever encoding a sequence is beneficial, we take care to exclude its sub-sequences from the histogram: for example, when encoding an equality comparison and a branch together, we update the histogram not to count the sequence as occurrences of either a comparison or a branch. Figure 4-3 shows the effect of maximal microcoded sequence length on overall instruction density. The multi-cycle scheme improves on the single-cycle ideal scheme in that the average bit width is reduced to 4-7 bits per instruction, but maximum bit width increases to 15. A unit sequence length ($L = 1$) is identical to the ideal single-cycle encoding analyzed in the previous exercise. The ideal multi-cycle scheme is only given as lower bound on instruction bandwidth, as this approach requires tremendous decoder complexity, likely requiring large look-up table and a capability to fetch and decode misaligned instructions across cache lines.

4.3 Encoding the EM^2 ISA

Ideally, we would like to encode the EM^2 instruction set using an encoding function that represents all programs of interest favorably, and yields a simple and high-

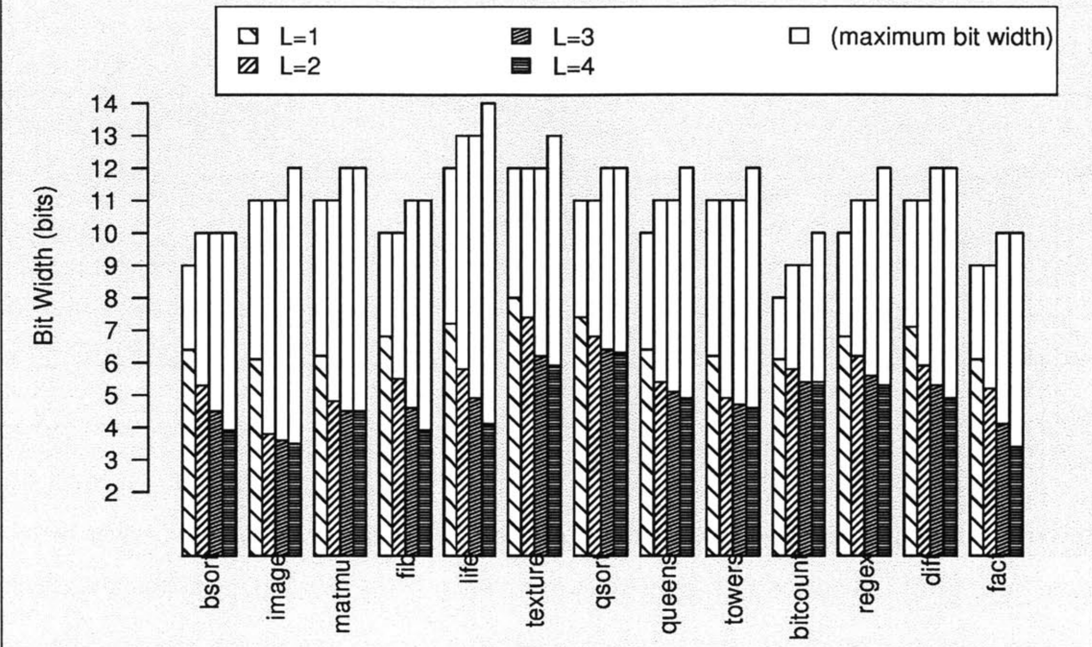


Figure 4-3: Average and maximum instruction widths with ideal multi-cycle encoding

performance decoder. To do so, we propose and discuss several encoding schemes, evaluating them in much the same way as the ideal scheme was evaluated earlier in this chapter.

4.3.1 Naive 32-bit Encoding

Much like in a RISC-like instruction set [28], we designate the various fields in each instruction to a set of bits in a 32-bit word. More specifically, we partition the word as shown in Figure 4-4, assigning the 16-bit immediate field used by instruction-coded constants and PC-relative control flow operators. A full word of data is more than enough to encode all instructions in this manner, meaning the decoder logic is minimal. This a safe and conservative (albeit inefficient encoding), and is the encoding implemented in our chip [36].

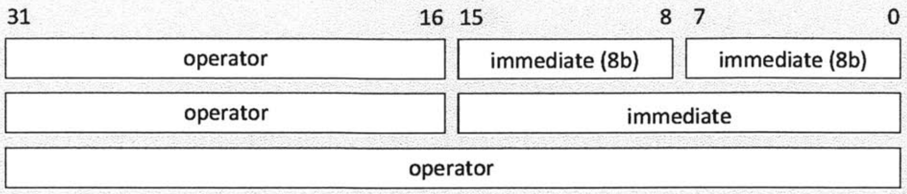


Figure 4-4: Naive 32-bit instruction encoding for EM^2

4.3.2 Dense 8-bit Encoding

To better match the largely sparse nature of the EM^2 instruction set, we create a byte-sized encoding. Because 8 bits is insufficient to encode the immediate fields of several instructions, we change the semantics of these operators to read their immediate values from the hardware stack. This makes constants somewhat problematic, requiring us to use eight separate instructions to initialize a 32-bit constant. To be consistent, we count all overhead instructions introduced in this manner as a single, larger instruction, meaning a constant requires 64 bits of the instruction stream. Further overhead is actually incurred due to underutilized datapath cycles. The instruction encoding is very dense, meaning the decoder is a dense two-level circuit of combinational logic, though with good per-instruction performance. This encoding is similar to those used by ultra low-power stack and accumulator-based microcontrollers.

4.3.3 Compressible Encoding

While the 32-bit scheme densely encodes a few of the EM^2 instructions (precisely the instructions requiring a large immediate value), most operators do not require such a wide instruction word. Conversely, while the 8-bit encoding captures much of the ISA fairly well, it suffers greatly from overhead incurred when splitting long instructions into inefficient instruction sequences.

We combine the two schemes via a compressible encoding [25]: extending a naive 32-bit encoding (truncated to 31 bits in this case) with tuples of 4 instructions encoded as a single word. Each instruction in the tuple is represented using a byte (except the first, which only uses 7 because the decoder must differentiate compressed and uncompressed instructions). By employing a variable width encoding, we take advantage of the largely sparse ISA by encoding it densely, while accommodating instructions with big immediate values. Typically, a variable instruction width introduces a slew of difficult problems. Firstly, by allowing instructions to be encoded as non-uniform sequences of bits, instruction alignment becomes non-uniform as well, meaning an executable may maliciously or erroneously transfer control to an address that does not correspond to the start of an instruction. The decoder happily translates such control flow transfers as a new instruction stream, which has a completely different meaning than the first, making variable width programs extremely difficult to analyze and secure. Another issue occurs when long instructions are not aligned to cache line boundaries: if an instruction resides in two separate cache lines at the same time, additional hardware must be implemented to handle the odd case of missing two cache lines simultaneously. We alleviate this problem by compressing tuples of 4 (or fewer) instructions into a single word, or "super instruction". Branch targets, as well as all encoded instructions, must align to word boundaries, meaning each instruction belongs to precisely one cache line.

Only a subset of the ISA may be compressed in this manner: 7-8 bits is insufficient to represent the entire ISA, as discussed in Section 4.3.2. Realizing that this scheme is a limited case of a "super instruction" 32-bit encoding with a maximum sequence length of 4, we use the same frequency analysis technique used to compute ideal multi-cycle encoding to derive the set of instrucitons used most frequently. 128 of these common opcodes are encodable as the head byte of the 4-instruction tuple. The other three may use the full 8 bits, for a total of 256 compressed instructions. The instruction encoding is illustrated in Figure 4-5. We take care to separate the "training set" and the evaluation benchmark. The compressed instructions used when executing each benchmark are computed from the union of all other benchmarks, excluding the test program, as is done in experiments utilizing cross-validation [24].

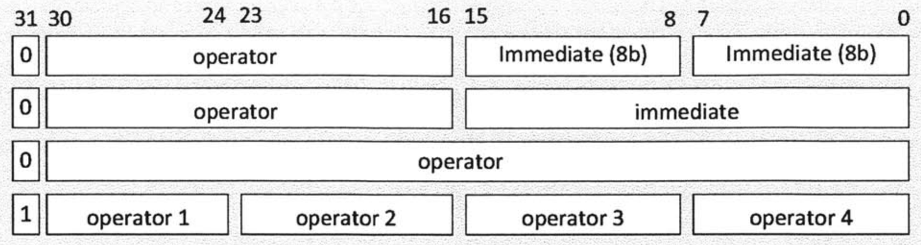


Figure 4-5: Compressible instruction encoding for EM^2

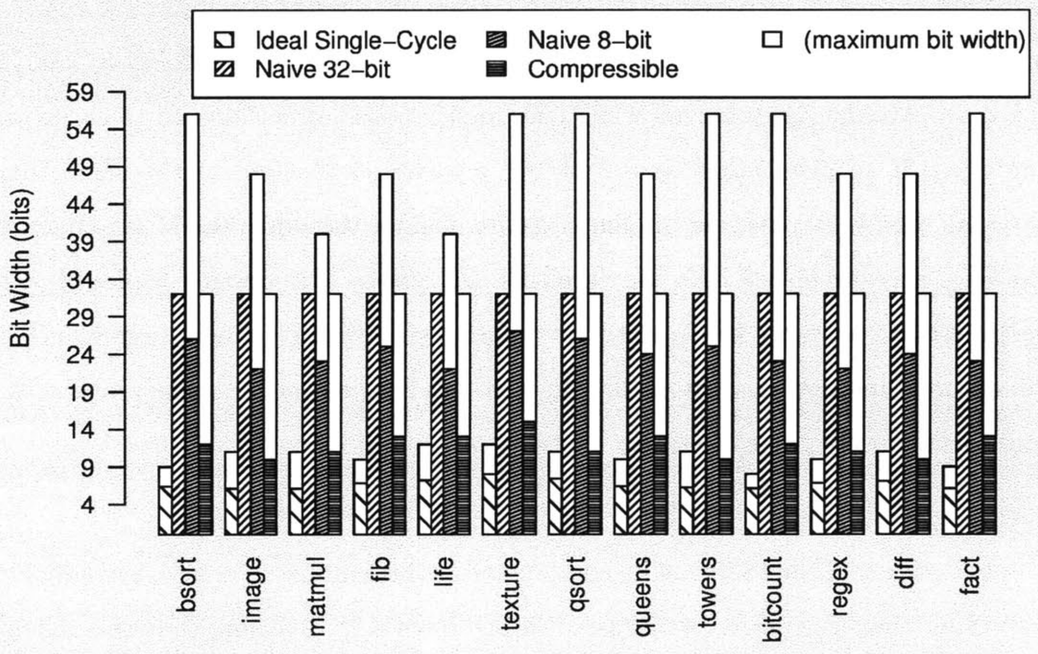


Figure 4-6: Average and maximum bit widths for surveyed EM^2 instruction encoding schemes

4.3.4 Multi-Cycle Compressible Encoding

By extending the already dense compressible encoding scheme with "super instructions" we initially attempted to further increase the efficiency of the compressible encoding scheme. We pursued two different schemes: super-instructions encoding a sequence of 4 without individually decodable members (requiring a table lookup to translate the 32-bit word into 4 individual instructions), and a separate scheme allowing longer sequences to be encoded as a 32-bit word. Both schemes suffered from high decoder complexity, and observed seemingly irrelevant sequences appearing frequently. After further investigation, it became clear that much larger program traces would need to be surveyed to produce a representative encoding with large sets of "super instructions", especially if long sequences are permitted. Neither scheme is included in the evaluation.

4.4 Evaluation of ISA Encoding Schemes

We evaluate the encoding schemes by compiling a set of representative benchmarks, and executing them on an ISA simulator for the EM^2 architecture. We then post-process the benchmark traces to analyze the average and maximum bit width of the instructions evaluated in each trace. Specifically, we evaluate the naive sparse (32-bit) encoding, the naive dense (8-bit) encoding, as well as the compressible encoding. We use the ideal single-cycle encoding as a lower bound to illustrate the relative efficiency of each encoding scheme. The results are shown in Figure 4-6. The naive sparse scheme fails to take advantage of the sparse nature of the EM^2 ISA, both its average and maximum instruction bit width are trivially 32 bits, well in excess of the ideal scheme. The naive dense scheme performed relatively poorly as well: although each encoded instruction is exactly 8 bits wide, many instructions were split into instruction sequences, as the encoding does not permit large immediate fields required by the EM^2 instruction set. These sequences were counted as a single, wide instruction, raising both the maximum and average bit width of the naive dense scheme to 56 and 27 bits, respectively. The `texture` benchmark showed the naive dense encoding scheme to be particularly inefficient due to numerous control flow instructions and offset memory accesses: all instructions requiring a lengthy sequence to implement the immediate offset. The compressible encoding scheme performed relatively well, with an average bit width of 12 bits, approaching ideal efficiency. Maximum bit width for this scheme is trivially 32 bits, as the encoding scheme is identical to the naive sparse scheme in its uncompressed form.

4.5 Conclusion

In this chapter, we discussed the problem of encoding the EM^2 instruction set. Because the ISA is stack-based, it suffers from overhead in form of additional instructions needed to explicitly manage the hardware stack, contributing to a larger code size. To mitigate this problem, we observe that the ISA is quite sparse, and devise a dense encoding to compress the instruction stream, thereby reducing code size significantly. Specifically, we discuss the problem of encoding an ISA in abstract terms, and derive an upper bound for instruction density for use as a baseline for evaluating candidate encoding schemes. We then propose several schemes to encode the EM^2 instruction set, and demonstrate that a compressible ISA scheme performs best among the methods surveyed. We show that this encoding approaches ideal instruction density, but does not require a stateful or otherwise complex, slow decoder

A dense instruction encoding increases effective instruction cache size (in that a large number of instructions can be cached in each line of the cache), decreasing miss rate and network bandwidth required. Another benefit is that a dense encoding negates code size overhead from explicit stack management produced by variable scheduling, as discussed in Chapter 3. While variable scheduling expands the instruction stream by adding 0.33 stack operations per instruction on average, the compressible encoding scheme reduces the code size to 37.5% of a simple 32-bit encoding used by many RISC processors. As a result, the code size of a EM^2 program incurs only 12% overhead due to stack variable scheduling relative to a naive 32-bit encoding.

Chapter 5

Library for Synchronization in the EM^2 Execution Model

In previous chapters, we designed and evaluated subsystems of a compiler capable of transforming architecture-agnostic C code into an executable program for the EM^2 architecture. The compiler is optimized for the EM^2 execution model and produces relatively efficient machine code across a range of representative single-threaded benchmarks. Despite this result, the EM^2 execution model must capture the machine's key characteristics, chief among them being that EM^2 is by design a highly parallel machine. With over a hundred threads of execution united by a scalable shared memory abstraction, we must provide affordances to allow the programmer to describe parallel workloads. Unfortunately, because our platform fundamentally differs from well-established computer systems, we found no existing libraries that easily conform to the EM^2 architecture. This thesis makes no attempt to solve the parallelizing compiler problem in the general, and instead delegates the parallel software problem to the programmer by providing a set of interfaces and library primitives to enable thread creation, communication, and synchronization.

In this chapter, we design and evaluate a basic set of synchronization primitives optimized for the EM^2 architecture. Although synchronization primitives are only a small set of a rich application binary interface (ABI) for a multi-threaded environment, we use this discussion to illustrate the design methodology and optimization goals for library primitives in general. Unlike most library blocks, which simply expose an architectural feature and require straightforward engineering effort, performance and correctness of synchronization primitives is intricately tied to the on chip network and memory subsystems. We discuss how and why a naive implementation of a lock fails when using remote access for mutual exclusion. Using the insights gained from this discussion, we design and optimize a low-level lock, and a high-level low-overhead lock primitive. Finally, we discuss the multicast subroutine of a barrier implementation, and describe an optimization achieved by relaxing the ABI memory consistency model in a controlled way for the barrier implementation. Throughout the discussion, we demonstrate ways in which migration is an enabling technology for thread synchronization. We show how migration and remote access can be used together to optimize the performance, network utilization, and scalability of synchronization primitives on EM^2.

5.1 ABI Library

In Chapter 2, we introduced the ABI as a union of a language, a library, and a set of conventions that make them interoperable. The compiler aptly describes the language component of an execution model, exposing computation resources via the C programming language, but some useful constructs cannot be efficiently implemented on top of this abstraction. Such primitives are not excluded from a usable computer system; implementers break abstractions established for the programmer in controlled ways to provide a library written below the ABI abstraction, as shown in Figure 5-1. For example, thread creation and destruction is impossible in pure C, as the language has no concept of threads of execution. A programmer is able to use threads via a library, such as `pthreads` [11], implemented below the architecture-agnostic abstraction. By adhering to the same conventions as the compiler at library interfaces, implementers can create functions interoperable with compiled code, but have the option to deviate from the execution model.

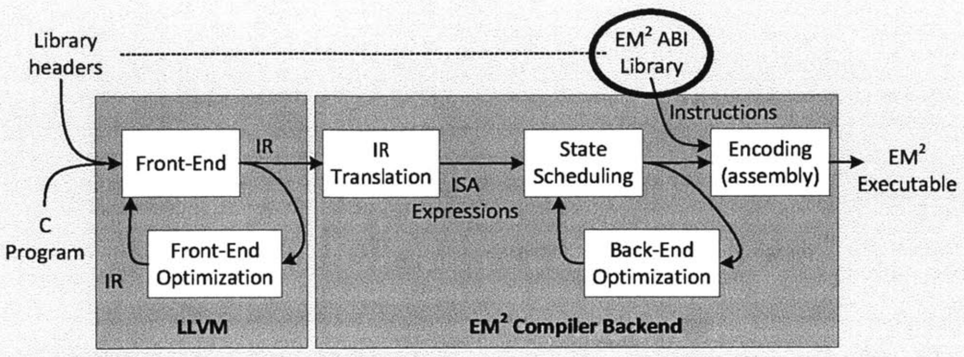


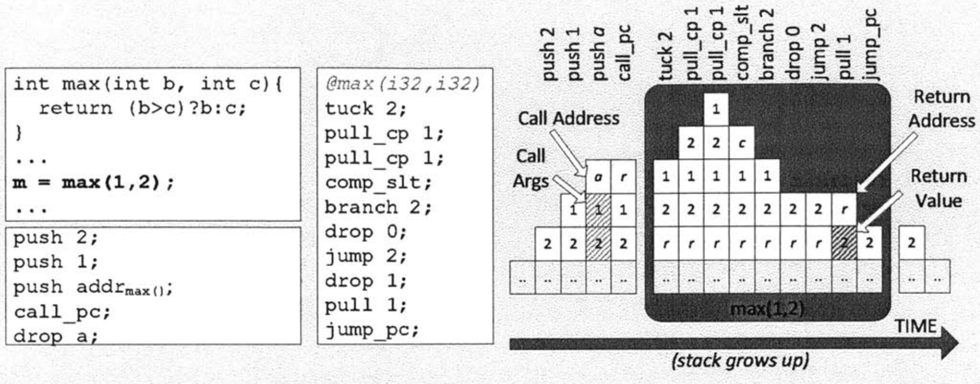
Figure 5-1: ABI library and its role in the compiler toolchain

5.1.1 Calling convention

Before we can create a useful library of primitives for the EM^2 architecture, we must discuss the EM^2 calling convention. Normally a function call creates a stack frame in, with function arguments passed as variables in the new stack frame, and return values passed as variables in the prior frame. Some architectures cache the variables in a register file to reduce function call overhead. The Execution Migration Machine has a hardware stack, making argument passing trivial: a function call can be exposed as an operator that takes arguments from the stack in argument order, and deposits any return values on the top of the stack also, much like a basic block expression. No interaction with the memory subsystem is necessary. This convention is illustrated in Figure 5-2. Both the library and the compiler implement this same calling convention, making the library interoperable with compiled programs.

5.2 Bit Movement Metric

The EM^2 project revolves around the idea of achieving a scalable shared memory abstraction by eliminating shared state, relying on local decisions to eliminate handshakes, and minimizing network traffic. The bit movement metric is a central concept in this discussion: movement of bits is arguably a good approximation for power effi-



```
int max(int b, int c){
  return (b>c)?b:c;
}
...
m = max(1,2);
...
```

```
push 2;
push 1;
push addrmax();
call_pc;
drop a;
```

```
@max(i32,i32)
tuck 2;
pull_cp 1;
pull_cp 1;
comp_slt;
branch 2;
drop 0;
jump 2;
drop 1;
pull 1;
jump_pc;
```

Figure 5-2: EM^2 calling convention

ciency (much of the expense of parallel workloads is incurred in the on-chip network due to communication and synchronization of threads), and helps illustrate scalability [13]. The criterion by which we optimize all library primitives is the same bit movement metric: by minimizing bits moved (network flits $*$ travel distance), we improve the scalability and latency (fewer blocking handshakes) of the primitives discussed in this chapter.

5.3 The Lock Synchronization Primitive

In a parallel program where multiple threads use memory concurrently, accesses to shared state are serialized and may result in ambiguity: when executed, a multithreaded program follows one of many possible interleavings of memory accesses, a source of non-determinism. In a massively parallel computer system such as EM^2, inter-thread communication is of particular importance because a large number of communicating threads creates countless interleavings, only a subset of which may describe correct behavior. Without a means to enforce a correct interleaving, data loss or even livelock may occur. To avoid this problem, we design a lock primitive [21] to allow programmers to implement mutual exclusion and explicitly orchestrate correct interleavings where a well-defined behavior is desirable for program correctness or performance.

```
Input: $A_L$ - address of a lock in memory;
$\exists$ a time such that $Mem[A_L] = 0$ (lock is logically "free").
Output: $Mem[A_L] = 1$ (Lock is logically "acquired").
while true do
    begin critical section
        if $Mem[A_L] = 0$ then
            $Mem[A_L] \leftarrow 1$;
            break;
        else
            //lock is not free. Retry
```

Algorithm 4: Naively acquire a lock primitive

```
Input: $A_L$ - address of a lock in memory;
$Mem[A_L] = 1$ (lock is logically "acquired").
Result: $Mem[A_L] = 0$ (The lock is logically "free").
begin
    $Mem[A_L] \leftarrow 0$;
```

Algorithm 5: Naively release a lock primitive

5.3.1 Naive Lock Primitive

A lock primitive is essentially an affordance for mutual exclusion given by two interfaces: `acquire()` and `free()`. The algorithm to acquire [21] a lock is given by Algorithm 4. To release a lock, no exclusion mechanism is needed: the entity releasing the lock must already own it, so exclusion is already guaranteed. The `free()` routine is given by Algorithm 5.3.1.

To implement the lock, we must implement a critical section, an atomic sequence. The EM^2 architecture allows atomics using a linked load (`fnc_ld_rsv`, among others) and a conditional store (`fnc_st_cond`, among others), as shown in Section 2.3. These instructions do not implement a read-modify-write sequence by themselves, and therefore do not execute as an uninterruptible sequence. Instead, `fnc_ld_rsv`

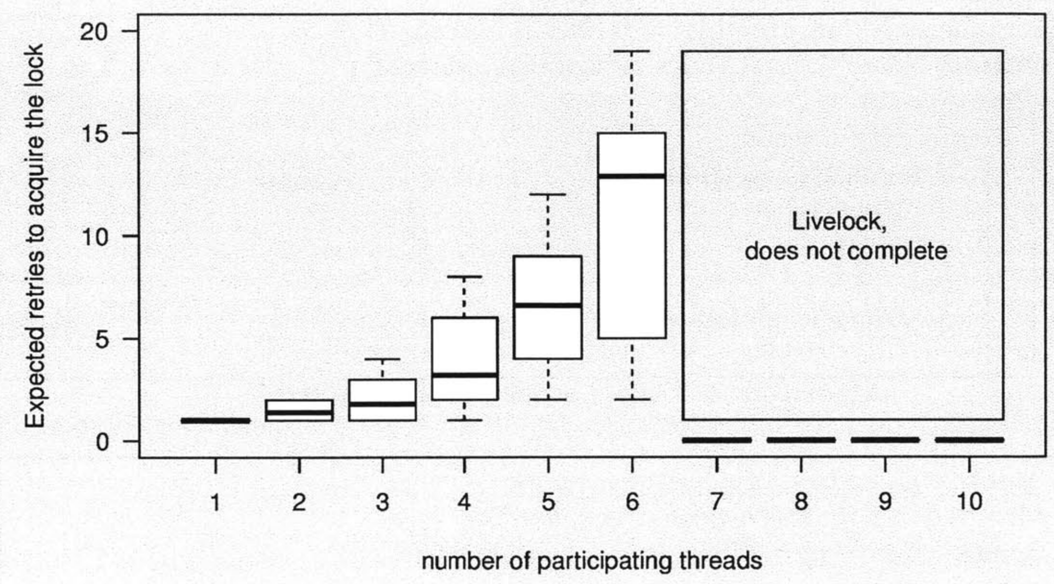


Figure 5-3: Livelock in naive lock implementation on EM^2

begins the sequence, and if any thread accesses memory before the write occurs, the store conditional instruction fails [26].

A critical section makes no forward progress if interrupted, meaning deadlock is possible if two threads compete for multiple resources creating circular dependency [37]. Worse yet, livelock may occur if the critical section is interrupted every time [57]. To demonstrate this behavior, we implement the lock primitive as shown above, and record the number of times each thread attempts a critical section to acquire the lock. Figure 5-3 demonstrates the naive lock implementation livelocks almost immediately with even a small number of competing threads due to reservation breaking.

5.3.2 Analysis of the Naive EM^2 Lock

To better understand the livelock behavior, we design an experiment to show how low-level behavior causes application-level livelock. The EM^2 architecture has two mechanisms for memory access: remote access and execution migration. The EM^2 hardware migration predictor [51] can dynamically select which of the two provide

```
Input: $A_L$ - address of a lock;
$\exists$ a time such that $Mem[A_L] = 0$
(lock is logically "free").
Output: $Mem[A_L] = 1$
(Lock is logically "acquired").
while TRUE do
    $L \xleftarrow{\text{fnc\_ld\_ra\_rsv}} Mem[A_L]$;
    if $L = 0$ then
        $Mem[A_L] \xleftarrow{\text{fnc\_st\_ra\_cond}} 1$;
        if fnc_st_ra_cond succeeds then
            break;
        else
            //interrupted, Retry
    else
        //lock is not free. Retry
```

Algorithm 6: Acquire a lock primitive via remote access (RA)

```
Input: $A_L$ - address of a lock;
$\exists$ a time such that $Mem[A_L] = 0$
(lock is logically "free").
Output: $Mem[A_L] = 1$
(Lock is logically "acquired").
while TRUE do
    begin unevictable critical section
        $L \xleftarrow{\text{fnc\_ld\_em\_rsv}} Mem[A_L]$;
        if $L = 0$ then
            $Mem[A_L] \xleftarrow{\text{fnc\_st\_em\_cond}} 1$;
            if fnc_st_ra_cond succeeds then
                break;
            else
                //interrupted, Retry
        else
            //lock is not free. Retry
```

Algorithm 7: Acquire a lock primitive via execution migration (EM)

better expected bit movement using simple reinforcement learning. For the purposes of this discussion, however, we separately examine two lock implementations: one using remote access only, and one using execution migration. Algorithms 6 and 7 shows the RA-only and EM-only lock algorithms side by side. The release routine is naive, and uses remote access, for both EM and RA locks because migration is obviously wasteful here. Algorithm 8 details the benchmark we use to evaluate each lock. Each thread acquires the lock one thousand times ($K = 1000$). C is minimized to maximize contention. The EM implementation forces each iteration to self-evict to the home core, preventing a thread from acquiring the lock multiple times after a single migration.

Figure 5-4 demonstrates the completion time for benchmark behavior for various numbers of threads contending for the same lock. Figure 5-5 demonstrates the bit movement associated with each benchmark. It is clear that the RA-only lock performs

```
Input: $A_L$ - address of a lock in memory;
$K$ - number of repeated trials;
$C$ - Critical section length;
for $i \in [1\ to\ K]$ do
    acquire($A_L$);
    wait(C);
    release($A_L$);
    self-evict;
```

Algorithm 8: Lock benchmark

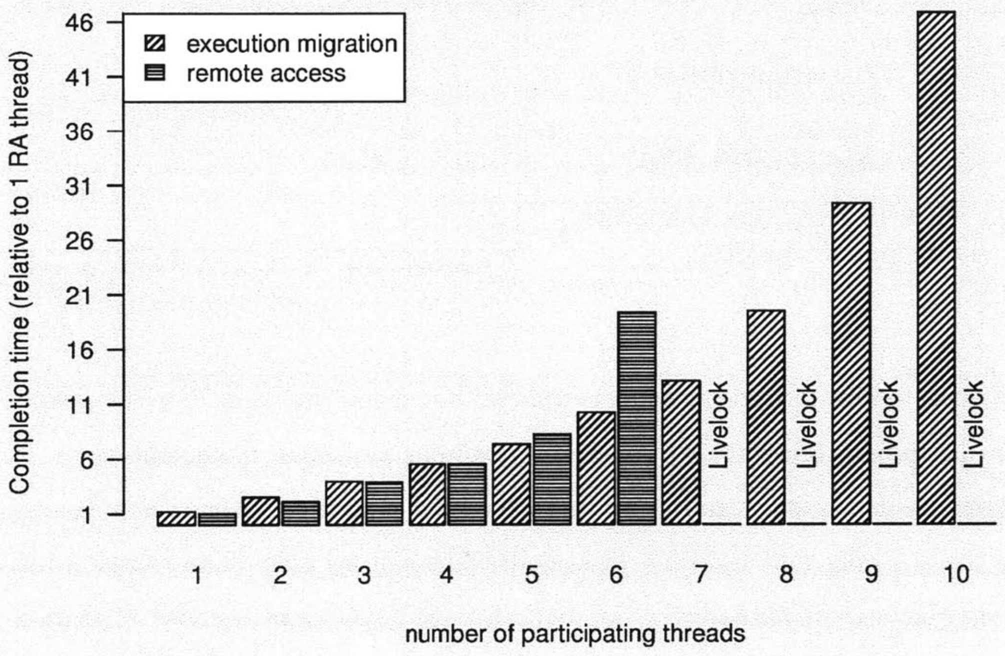


Figure 5-4: Completion time of benchmarks using EM-only and RA-only naive locks

better, both in terms of completion time and bit movement, than the EM-only lock when little or no contention is induced by the benchmark. This stands to reason because migration generates a larger network message than remote access, and incurs additional delay due to serialization and deserialization.

As the contention increases, however, the RA-only lock fails. The lock suffers from reservation breaking: increasing contention increases the rate with which threads attempt to acquire the lock, meaning the lock core is receiving `fnc_ld_ra_rsv` requests

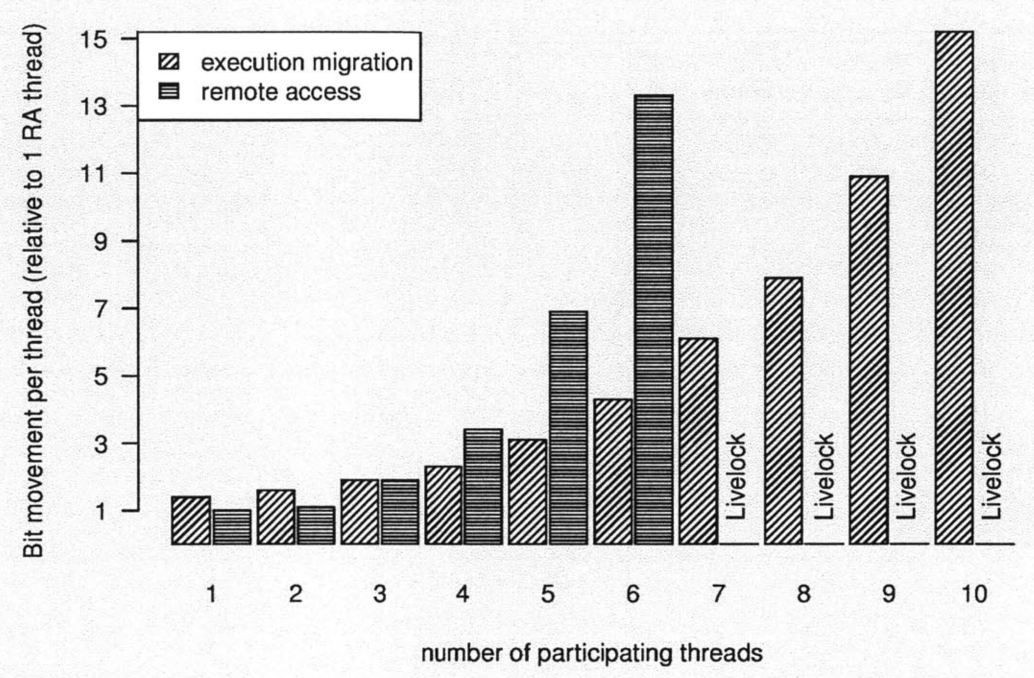


Figure 5-5: Bit movement observed for benchmarks using EM-only and RA-only naive locks (normalized to number of threads)

at a higher rate, breaking prior reservations and stalling forward progress. Livelock occurs when the expected time between remote load reserve requests becomes less than the critical section length, as shown in Figure 5-6 as a scatter plot of expected lock retries vs. expected time between load reserve requests. The critical section for the RA-based lock routine is 4 instructions long; livelock balloons when expected time between requests falls below 4 instruction commits.

Interestingly, the EM-only lock implementation appears to be resilient to the reservation breaking behavior. The cause is not immediately apparent: after a migration, the core hardware ensures several instructions commit in the guest context before allowing another guest to enter the core, evicting the 1^{st} visitor. A small C was chosen for the benchmark, making livelock impossible by ensuring the guest core executes the critical section entirely before allowing another guest to break the reservation. To better illustrate this behavior, consider Figure 5-7 where we vary C to show that a longer critical section can indeed be interrupted, causing livelock, albeit not as readily

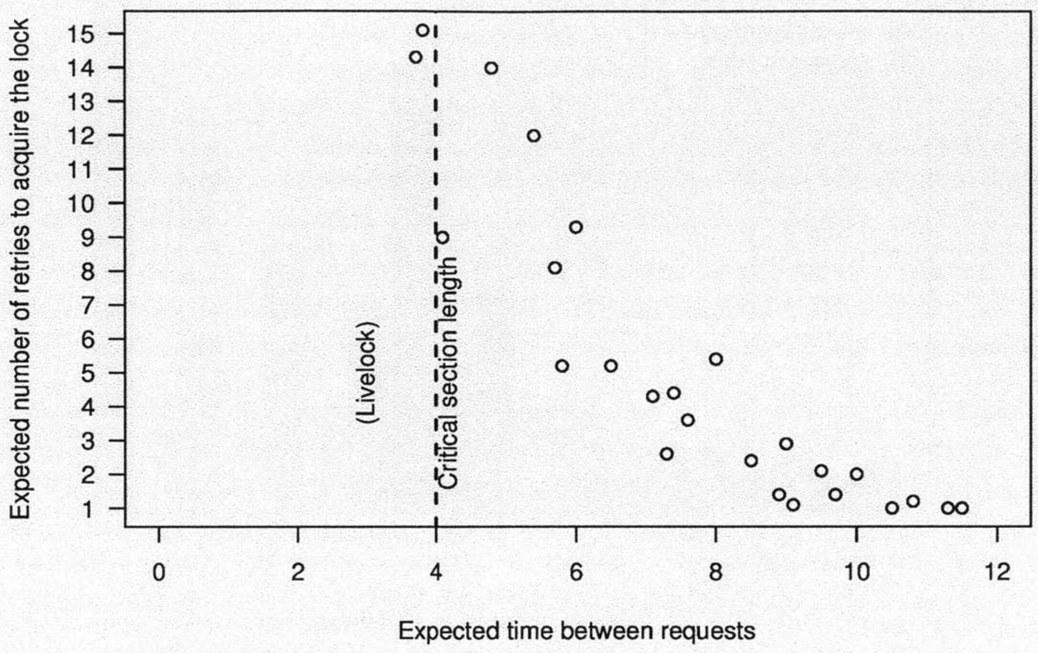


Figure 5-6: Livelock occurs when time between reservation-breaking requests is shorter than critical section length

as the in RA-only lock (due to the serialization latency limiting the load reserve rate in the lock core).

Because only a few variables are needed to complete the critical section, the EM-based lock benefits from a partial context migration. In fact only 3 values are sent, meaning the bit movement statistic is not much higher for the EM lock than for the RA lock - strong evidence that partial context migration is a far better option for mutual exclusion than RA. The guests attempting to acquire the lock are serialized on the network, however, potentially causing problems for other work using the network, and cannot execute until they enter the guest context. The hardware predictor, as utilized in the naive experiment at the beginning of this chapter, learns that remote access is most appropriate for the lock's free routine: the run length of the free routine is minimal, and the utility of a migration is very low, as only a small number of instructions can be executed on the guest core. Likewise, the predictor favors remote access for the acquire routine because, when successful, its run length is very

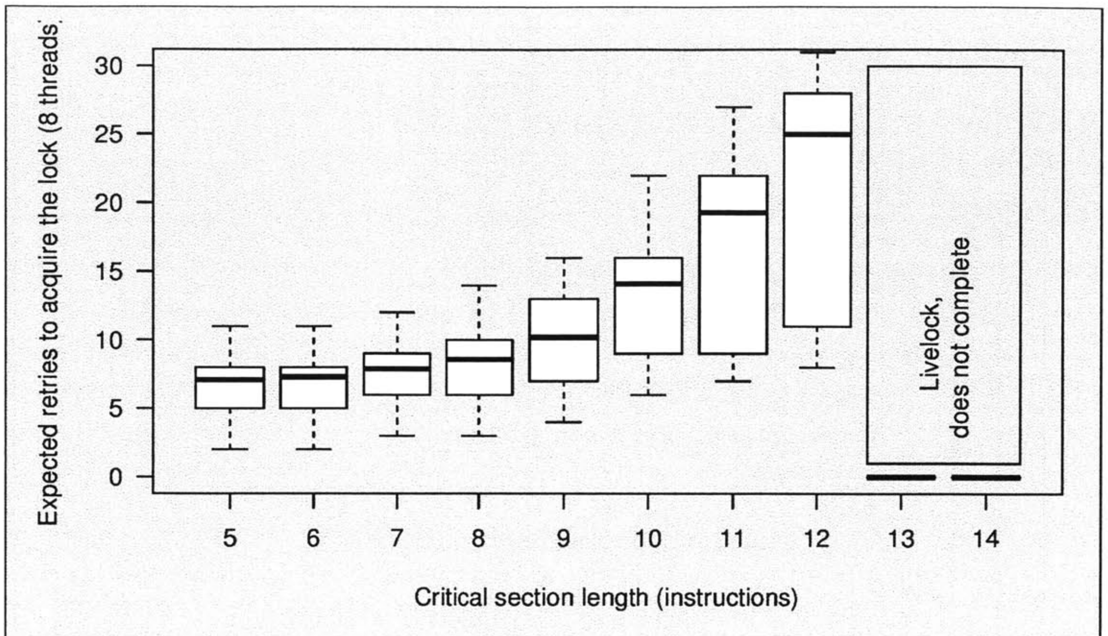


Figure 5-7: EM-only lock also livelocks if lock critical section is artificially elongated

short, meaning migration utility is low. Conversely, when the routine fails to acquire the lock, it generally gets evicted by another thread, and is ignored by the predictor's reinforcement learning mechanism.

5.3.3 EM-RA Hybrid Lock

Clearly, the EM-only lock implementation is far superior to the RA-only lock, as it is correct, and avoids deadlock. The EM lock does, however have a number of shortcomings, chief among them its heavy use of the on-chip network to arbitrate the lock. When the lock is owned, the threads trying to acquire the lock relentlessly migrate to the lock core, evicting each other frequently. Worse yet, contexts queue up in the network, potentially causing other problems.

To address these shortcomings, we would like to check whether a lock is available prior to initializing a migration. We issue an RA load, which checks whether the lock is free (without breaking an existing reservation if the lock is being acquired) before migrating, and forgo the migration altogether if the lock is owned. The lock acquire

```
Input: $A_L$ - address of a lock in memory;
$\exists$ a time such that $Mem[A_L] = 0$ (lock is logically "free").
Output: $Mem[A_L] = 1$ (Lock is logically "acquired").
while $TRUE$ do
    $L_{ra} \xleftarrow{\texttt{fnc\_ld\_ra}} Mem[A_L]$;
    if $L_{ra} = 0$ then
        begin unevictable critical section
            $L_{em} \xleftarrow{\texttt{fnc\_ld\_em\_rsv}} Mem[A_L]$;
            if $L_{em} = 0$ then
                $Mem[A_L] \xleftarrow{\texttt{fnc\_st\_em\_cond}} 1$;
                if fnc_st_em_cond succeeds then
                    break;
                else
                    //critical section was interrupted. Retry
            else
                //lock is no longer free after migration. Retry
    else
        //lock is not free. Retry
```
Algorithm 9: Acquire a lock primitive using a hybrid EM-RA routine

algorithm is given by Algorithm 9; while the free routine is naive and uses remote access, just as in our previous lock implementations.

The EM-RA hybrid lock dramatically improves network utilization, reducing bit movement, which serves to reduce lock contention and improve completion times when the lock is shared by a lot of threads, as shown in Figure 5-8. Figure 5-9 shows bit movement associated with the hybrid lock algorithm. The hybrid lock does have some overhead, requiring an additional handshake prior to migration, which makes it slower and more costly when few sharers are involved. Another consequence of the additional handshake is that when the lock is freed, no threads are queued on the network to acquire it. Only after a remote access handshake do threads attempt to migrate and acquire the lock, meaning the lock is free for a short period after each eviction. This is behavior is shown in Figure 5-10, which compares lock utilization (ratio of time the lock is free to time owned) and reveals that the hybrid lock forfeits some lock occupancy for its various benefits.

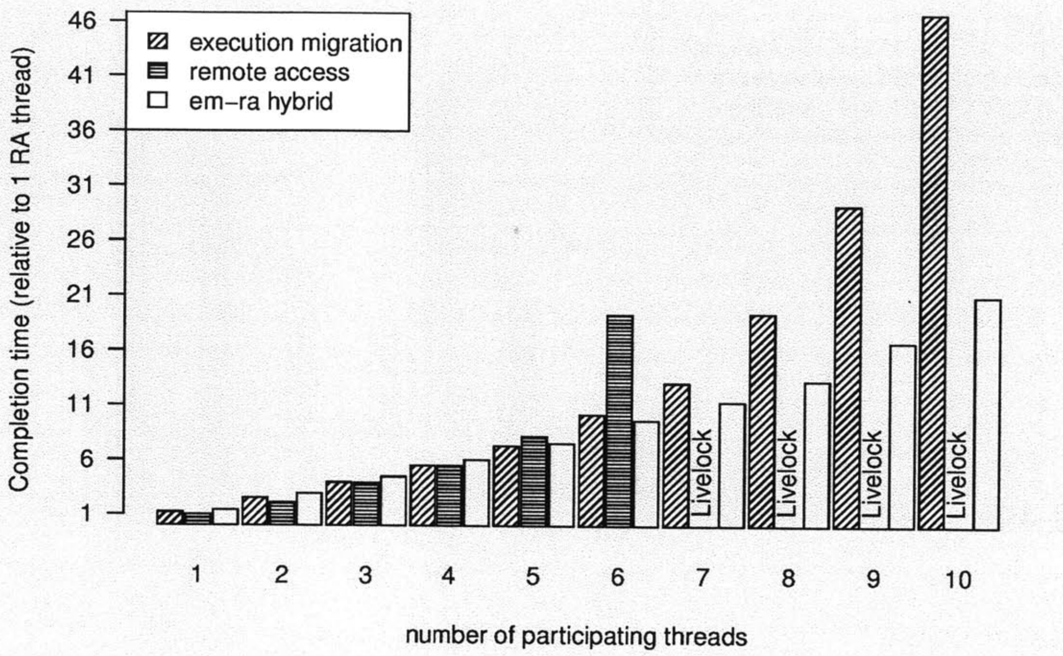


Figure 5-8: Completion time with RA, EM, and EM-RA locks

5.3.4 Self-Arbitrating Lock for Low Bandwidth

Taking the idea of minimizing bit movement further, we can address the retry behavior of the EM^2 lock. Although this optimization is not suitable for fast, fine-grained locks, it serves to dramatically reduce network traffic associated with the lock by offloading arbitration from the network to the lock core [20, 16].

The idea is to eliminate retries completely: when thread T_a attempts to acquire a lock already owned by thread T_b, it adds itself to a queue of waiting threads, then self-evicts to its home core and spins on a local lock (located in the home core's cache, therefore no network traffic is generated). When the lock is released by thread T_a, the releasing thread arbitrates the lock (selects the next thread in the queue, for example), and notifies the winning thread T_b that it won arbitration. The lock is never released - it is simply transferred from T_a to T_b, meaning lock utilization is

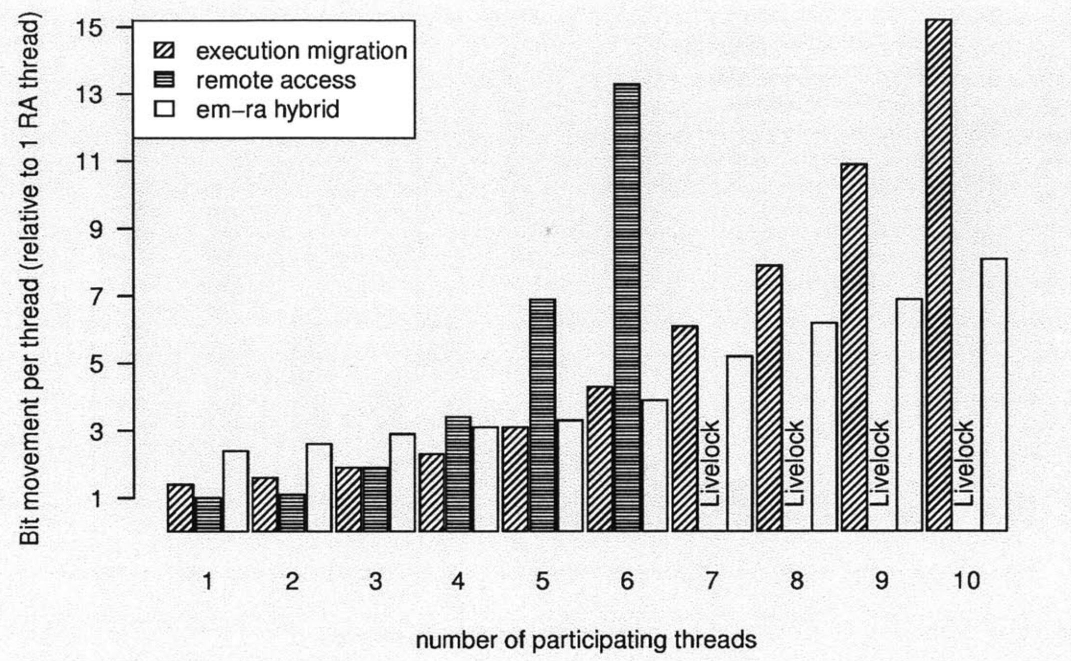


Figure 5-9: Average bit movement per thread observed with RA, EM, and EM-RA locks

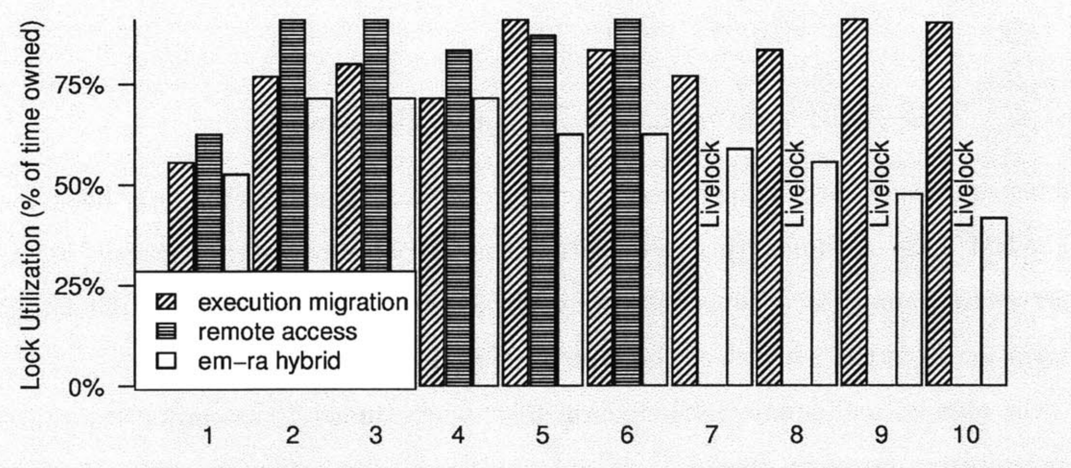


Figure 5-10: Lock utilization observed with RA, EM, and EM-RA locks

optimized also.

The lock data structure (the queue) must be guarded by a fine-grained lock, such

```
Input: $A_{guard}$ - address of a lock in memory;
$L$ - variable to represent lock state;
$Q$ - a shared queue, initially empty;
Output: thread has logically acquired the self-arbitrating lock.
begin
    acquire($A_{guard}$);
    if $L$ is "free" then
        $L \leftarrow$ "acquired";
        release($A_{guard}$);
    else
        Let $L_{sleep}$ be a new lock on the thread's home core.;
        acquire($L_{sleep}$);
        Q.append($L_{sleep}$);
        release($A_{guard}$);
        acquire($L_{sleep}$); // Sleep on the newly allocated lock at home core.
        // When $L_{sleep}$ is remotely released, the thread won arbitration, and
        owns $L$.
```

Algorithm 10: Acquire a self-arbitrating lock primitive

as the EM-RA lock. The algorithms for acquiring and freeing the self-arbitrating lock are shown in Algorithm 10 and Algorithm 11, respectively. Figure 5-11 and Figure 5-12 shows the completion time and bit movement, respectively, comparing the EM-RA lock and the self-arbitrating lock, as well as the other locks evaluated in this chapter. The self-arbitrating lock allows very high utilization in a high-contention scenario because it does not arbitrate the lock on the network. When a thread releases the lock, it is transferred to the next waiting thread (if any). The high utilization of self-arbitrating locks is shown in Figure 5-13. While the self-arbitrating lock is very efficient and scalable, it is relatively expensive (requires an EM-RA lock to guard the internal data structure). The guard lock theoretically may suffer from high contention, but this has not been observed in practice, and can be easily remedied by implementing an exponential back-off scheme [5], as the self-arbitrating lock is not well-suited for fine-grained mutual exclusion anyway.

```
Input: $A_{guard}$ - address of a lock in memory;
$L$ - variable to represent lock state;
$Q$ - a shared queue;
Output: thread has logically released the self-arbitrating lock.
If any threads were waiting, another wins arbitration and owns $L$.
begin
    acquire($A_{guard}$);
    if $Q$ is empty then
        $L \leftarrow$ "free";
    else
        Let $L_{sleep} \leftarrow Q.removeFirst()$;
        release($L_{sleep}$);
        // Another thread owns $L$; $L$ is not "freed" during the transaction.
    release($A_{guard}$);
```

Algorithm 11: Release a self-arbitrating lock primitive

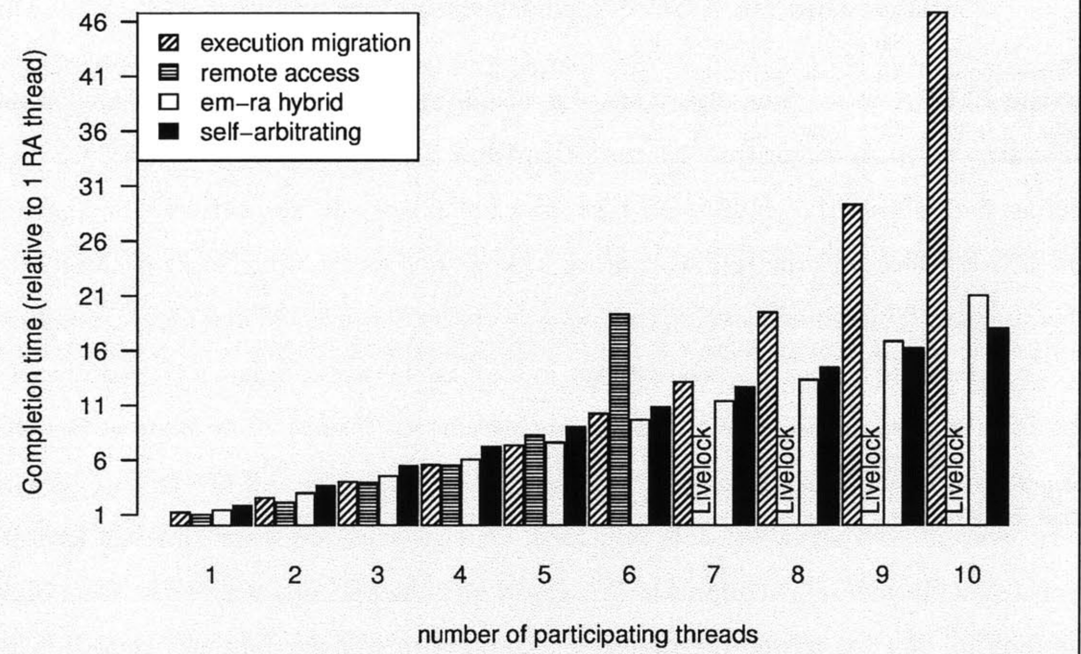


Figure 5-11: Completion time with self-arbitrating lock vs. other locks

5.4 The Barrier Synchronization Primitive

Although the lock primitive is sufficient to compose a rich synchronization library including message passing, semaphores, and other high-level primitives, the synchro-

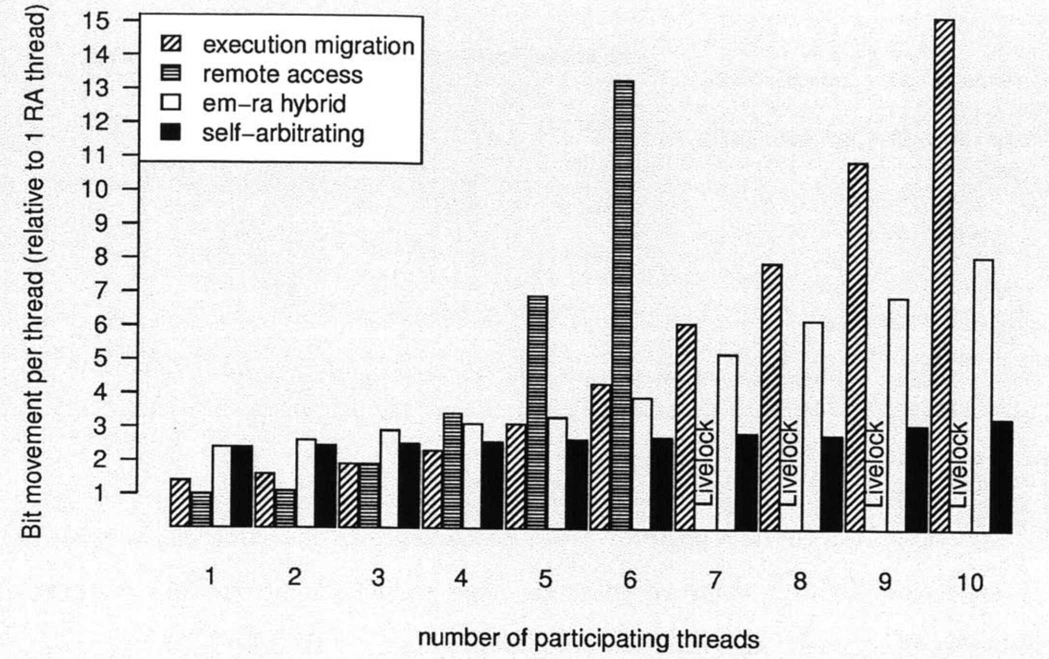


Figure 5-12: Average bit movement per thread observed with self-arbitrating lock vs. other locks

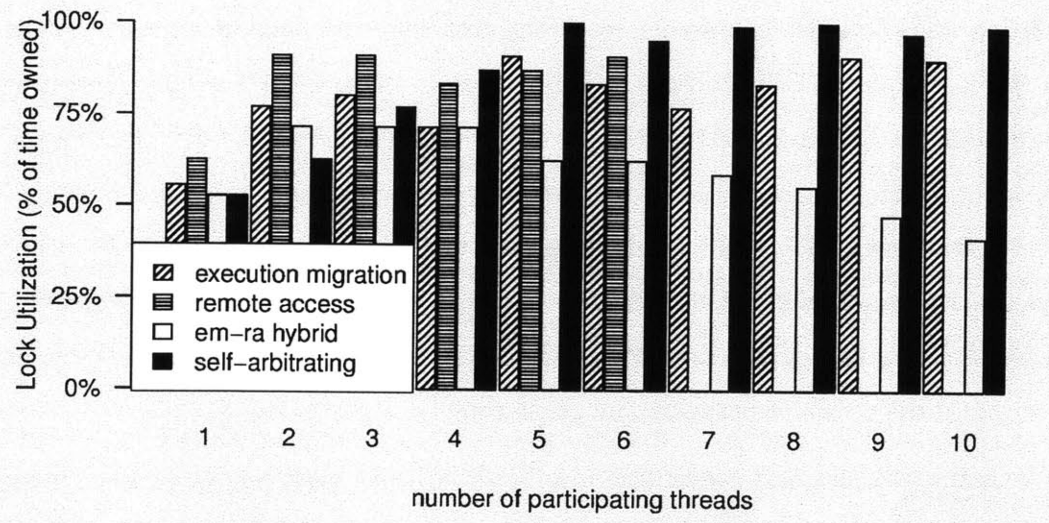
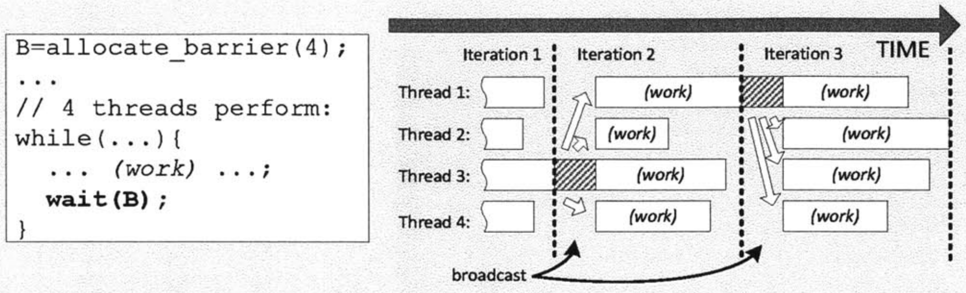


Figure 5-13: Lock utilization observed with self-arbitrating lock vs. other locks

nization barrier offers insightful architecture-specific optimization. A barrier [40, 52] is a synchronization tool used to orchestrate program "phases": the barrier ensures



```
B=allocate_barrier(4);
...
// 4 threads perform:
while(...){
   ... (work) ...;
   wait(B);
}
```

Figure 5-14: Synchronization barrier functionality

that all participating threads perform work associated with the same phase, preventing threads from pulling ahead of the rest. The barrier's main routine is `wait()`, which blocks the thread until all other threads have cleared the barrier, as illustrated in Figure 5-14.

The main challenge in implementing the barrier is its multicast mechanism: when all but one thread have reached the barrier, and are now blocked, the last thread executing `wait()` must communicate to the rest that the barrier is clear. While this seems simple, ENC [13] (the on chip network of the EM^2) makes numerous design decisions to eliminate the need for broadcast-like messages. For this reason, implementing a multicast on EM^2 is relatively expensive: multiple individual messages must be sent, each with a handshake, meaning multicast is serialized as a sequence of round-trip handshakes. This behavior is enforced in the ABI to ensure memory accesses are core consistent: memory stores made by a single core are observable as occurring in their correct order by any other core.

We can break the ABI convention in a localized, controlled way to to implement an efficient multicast: by deviating from a safe memory consistency model, we avoid a sequence of blocking memory accesses, and instead issue a number of non-blocking requests, then wait for the round-trip handshakes to complete in parallel. By observing that the order in which a multicast message is delivered to its recipients is irrelevant, we can defer the handshake and avoid serializing messages comprising the

```
Input: $A_{guard}$ - address of a lock in memory;
$N$ - an integer number of threads participating in the barrier;
$\{L_t\}$ - one local lock for each participating thread, initially “owned”;
$B$ - an integer initialized to $N$;
Output: routine returns only after all participating threads have called it.
acquire($A_{guard}$);
if $B > 1$ then
    $B = B - 1$;
    release($A_{guard}$);
    acquire($L_t$); //Sleep on local lock $L_t$
else
    // wake all waiting threads
    for $t_{remote} \in$ { other participating threads } do
        $Mem[L_t] \xleftarrow{\texttt{st\_ra\_noack}} 0$; //Non-blocking messages
    $B = N$;
    release($A_{guard}$);
```

Algorithm 12: Efficiently synchronize multiple threads of execution

multicast [60]. The barrier Algorithm is now relatively simple, and is given by Algorithm 12. Local locks are essential for the barrier to perform well, as busy waiting on the L_t locks does not generate network traffic. This barrier implementation is advantageous compared with a barrier on a directory-based architecture: only one broadcast is sent when the barrier is cleared, as opposed to a broadcast to update the barrier semaphore once per thread on a directory-based machine.

To evaluate the barrier, we benchmark a short loop and block on each iteration. We vary the number of participating threads, and compare the efficient barrier implemented above with a naive barrier (which uses a series of blocking messages by repeatedly calling `free(`L_t`)` instead of a non-blocking multicast implemented by `st_ra_noack`. The completion time is shown in Figure 5-15 relative to performance of a single participating thread, and demonstrates the dramatic improvement in both performance and network utilization brought about by an efficient multicast. To further prove the concept of multicast messages, we evaluate a multicast with N recipients for $N \in Z_{10}$ on an RTL implementation of a 110-core EM^2 chip. The mul-

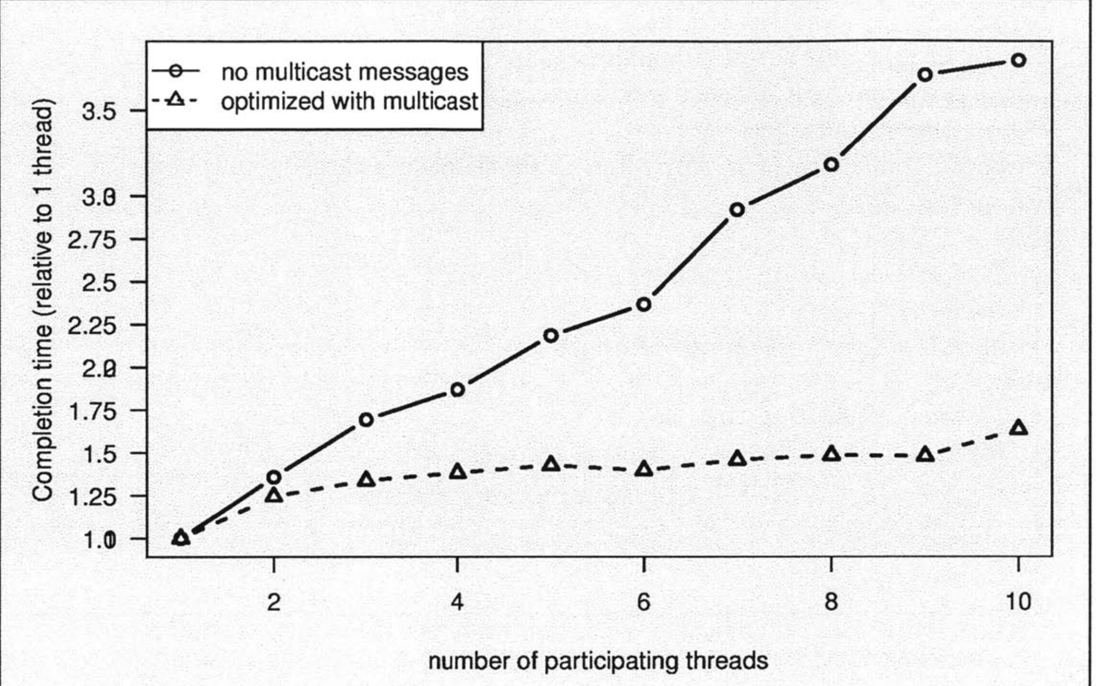


Figure 5-15: Completion time with optimized vs. naive barrier

ticast scales as expected; completion times are plotted in Figure 5-16. Bit movement is not significantly affected by the multicast implementation.

5.5 Conclusion

In this chapter, we use a bit movement metric introduced in related work for the EM^2 project as an optimization guideline throughout the chapter to study and implement highly optimized synchronization primitives for the execution migration platform. Specifically, we defined a calling convention and used it to implement a simple library primitive for mutual exclusion (a lock). We then demonstrated livelock behavior resulting from naive use of remote access atomics for mutual exclusion, and analyze the behavior of various atomic operators available on EM^2 in the context of a simple lock. After benchmarking the simple lock primitive and reasoning about its shortcomings in abstract terms, we use the insights gained to create a hybrid EM-RA algorithm for a mutual exclusion lock, and show that not only does it not livelock, but also performs

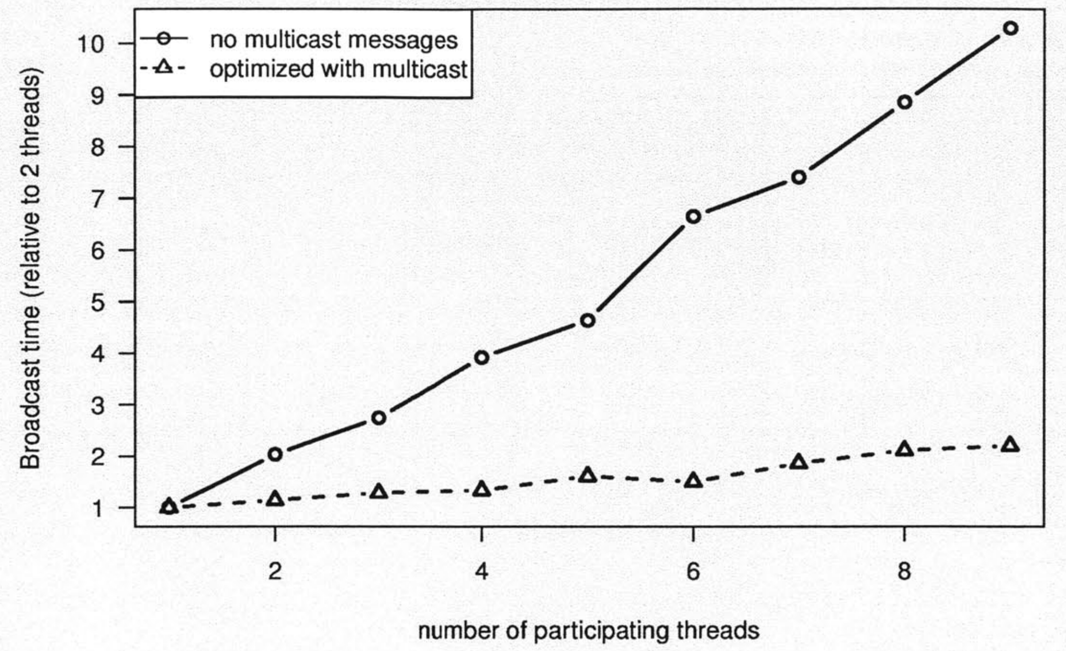


Figure 5-16: Multicast time with optimized vs. naive implementation, evaluated in EM^2 RTL

and scales well, both in terms of wall time and network utilization (bit movement). By further addressing the busy retry behavior of fine-grained locks, we describe a self-arbitrating lock optimized for the EM^2 execution model. The self-arbitrating lock, albeit too expensive for fine-grained exclusion, exhibits excellent scaling and performance for coarse-grained locks. Finally, we examine a barrier synchronization primitive, observe that its performance is hindered by a broadcast sub-routine (exactly the operation avoided by design at all levels of the EM^2 architecture) and design an efficient multicast technique by breaking the memory consistency provided by the ABI in a localized, and controlled way. Throughout the chapter we discuss execution migration, and partial context migration specifically, as an enabling technology for efficient thread communication and synchronization: threads are serialized onto a core by migration, providing a degree of mutual exclusion at the network level, making exclusion relatively inexpensive to implement in EM^2. We also definitively show that partial context migration is more adept for mutual exclusion than pure remote access, and discuss local locks for efficient thread sleep requiring no network bandwidth.

Chapter 6

Compiler Optimization for EM^2

The compiler problem in general consists of numerous sub-problems, many of which are not unique to EM^2. Architectural variation exists among compiler targets, but does not affect how high-level optimizations such as constant propagation are done. Furthermore, a compiler capable of lowering generic software to a specific target architecture is a monumental engineering effort, and is not generally well-suited for an academic project. Realizing this, we made an effort not to tackle the grand project of designing a compiler in its entirety. Instead, we rely on a well-established compiler infrastructure (LLVM) for front-end translation, high-level optimization, and analysis. In Chapters 2 to 4, we detailed the design of a custom compiler back-end to extend the LLVM framework and allow a programmer to target the Execution Migration Machine using C. Much of the effort discussed so far consisted of mapping execution to our unorthodox stack architecture, the subject of Chapters 2 to 4, but we have not yet discussed the high-level goal of compiling parallel workloads for the machine. While the compiler is correct, it does not optimize for the multi-threaded nature of most programs we expect to run. Although the ABI library touched upon in Chapter 5 provides many programmer interfaces for thread management, communication and synchronization, it remains true that the compiler is unaware of, and fails to optimize for key aspects of the machine, most notably execution migration, variable migration context size, and limited guest context size.

In this chapter, we design several classes of compiler backend transformations

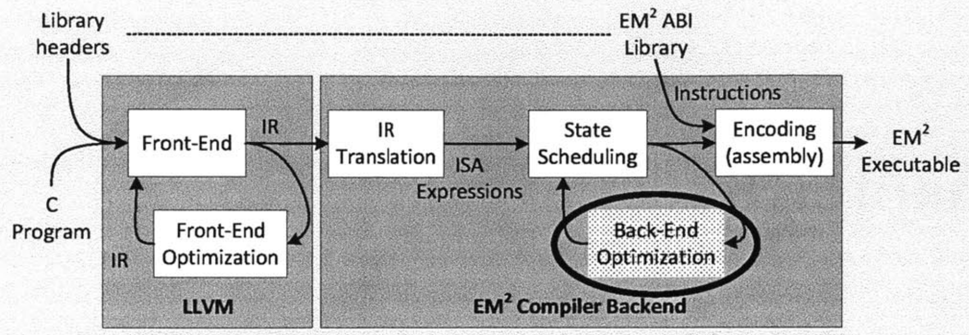


Figure 6-1: High-level role of back-end optimization in the EM^2 compiler

to improve performance and efficiency of compiled code on EM^2. Figure 6-1 shows the role of EM^2-specific optimization in the context of the entire compiler backend. Before we begin the discussion of specific techniques, we formulate our optimization goals, describe our parallel benchmarks, and discuss sources of possible inefficiency and propose two areas of optimization to improve compiled programs: migration and code size. Specifically, we analyze causes of migrations, and show that the majority of observed core misses are not beneficial, and should be eliminated wherever possible. We also discuss why code size reduction serves our optimization goals in multiple ways. Applying insight gained, we design and discuss several compiler optimizations to improve both migration efficiency and code size, ranging from simple transformations using the commutative property of some operators to complex rewrite passes to move many variable allocations out of memory onto the hardware stack. We also add a last-level peephole optimization to apply localized, fine-grained transformations to already compiled code. Finally, we evaluate several combinations of these passes and show that we are able to significantly improve both completion time, and bit movement for all benchmarks surveyed.

6.1 EM^2 Compiler Optimization Goals

In order to inform any optimization effort, we must discuss the metric(s) by which we gauge quality of resulting code. Performance in terms of wall clock time is an obvious measure by which we may judge optimality, as is the bit movement metric introduced in Section 5.2. Unfortunately, both metrics require a trace to measure, meaning the program must be executed to evaluate either. Not only is program evaluation impractically time-consuming in the context of a compiler, the results are highly data-dependent (for example, `factorial(1)` evaluates much faster than `factorial(100)`) and therefore requires information not available statically at compile time. We must formulate simpler optimization metrics as heuristics for measures of actual performance. To do so, we investigate major areas of improvement by which we can expect the compiler to affect program execution, and discuss our optimization goals in low-level terms. Specifically, we investigate migration behavior, which affects network bandwidth and overall program efficiency, and code size, which affects performance of individual threads, instruction cache use, and run lengths of migrations.

6.1.1 Migration Behavior

Migration plays a central role in program performance on the EM^2 system. Remote access and EM provide a shared memory abstraction, and all memory accesses use one of either mechanisms. Even for single-threaded programs, not all memory accesses are local if the program touches memory not native to its home core - a result of careless data placement or a working set large enough to exceed the core's memory space. Although execution migration potentially offers significant improvements in network utilization, and performance, it must be used judiciously to do so, as it also carries a significant overhead. Frivolous migration easily increases network congestion and thread idle time, decreases core availability, and may result in ping pong eviction patterns, all contributing to overhead in both performance and bit movement in major ways.

An ideal compiler for the EM^2 platform would be able to infer high-level program structure from C and programmer input, and optimize migration behavior accordingly. Unfortunately such work remains an open and challenging research problem, so we must rely on a set of heuristics to inform program optimization where migration is concerned. Simply speaking, our goal is to improve the utility of migration in the compiled program, which we restrict to increasing the utility of *each* migration (whole-program optimization with high-level goals lies well outside the scope of this thesis). We do not try to discover opportunities for additional execution migration via data placement optimization [43] or parallelization [22], and instead modify the migrations induced by the program's memory access as it is described by the programmer. In light of this, we seek to promote "good" migration (improves overall performance or bit movement) by reducing "bad" migration (introduces excessive overhead) by eliminating remote memory accesses wherever possible. To do this, we look into the causes of execution migration, which are enumerated in Table 6.1.

The relative frequency of each type of migration is shown in Figure 6-2 for a small set of non-optimized benchmarks. While some core misses may be beneficial to program efficiency, many occur because the program accesses the software stack. For the benchmarks surveyed, more than 70% of core misses were due to stack frame accesses - implicit state stored in memory (these are shown on the figure as `Native Core Miss (implicit)`). Conflict evictions are a result of other migration types, and can only be addressed by decreasing contention system-wide by the rate of migration. The other migrations, namely evictions due to underflow and overflow, which collectively make up the vast majority of all observed evictions, are strictly overhead and only decrease program efficiency. The compiler must eliminate these wherever possible. A run length heuristic [51] captures this optimization goal well: for each migration, we want to maximize the number of instructions executed on the guest core before a core miss or an eviction. The compiler's metric for optimization is simple: we want to reduce or eliminate potential for core misses (which occur at memory instructions), underflows and overflows (rearranging instructions to increase run length).

Migration type	native/guest	description
core miss	native, guest	memory access to a location not local to the current core
eviction	guest	another thread evicts the current thread due to a core miss
underflow	guest	the thread self-evicts because it needs data from the stack not available in the guest context
overflow	guest	the thread self-evicts because the guest context does not have enough free space on the guest stack to execute an instruction

Table 6.1: EM^2 migration types and corresponding causes

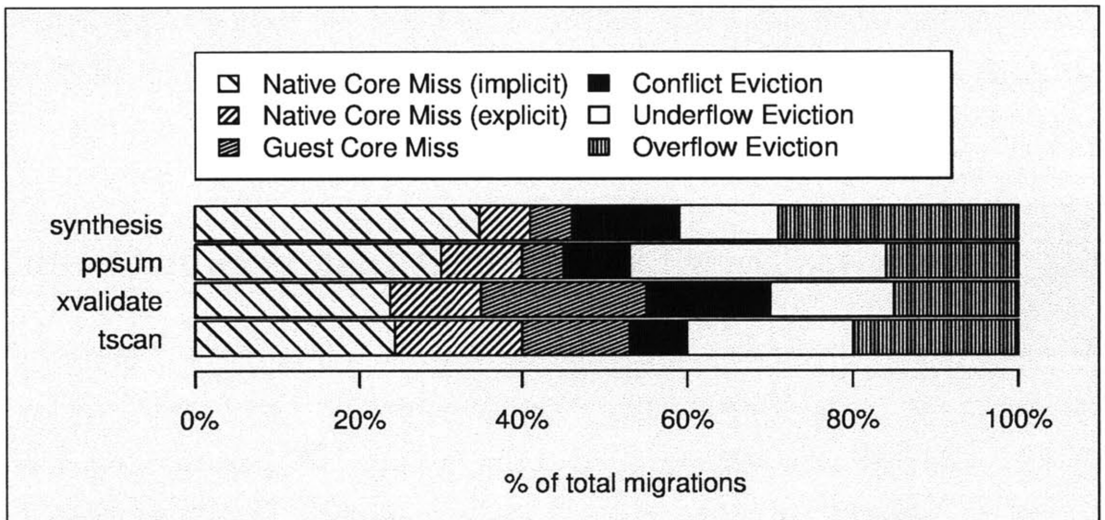


Figure 6-2: Relative frequency of migration sub-types on EM^2

6.1.2 Code Size

We previously discussed the issue of code size at length in Chapters 2 to 4. Excessive code size affects program performance by increasing network congestion and instruction cache miss rate. More specifically, excessive instruction count increases variable scheduling overhead and hardware stack access depth, and therefore decreases run length of migrations due to stack overflows and underflows. By reducing instruction count, the compiler would effectively both reduce total work to execute a program, and improve the program's migration and network bandwidth utilization at the same time. The heuristic for code size optimization is trivial: each instruction removed from the compiled program improves performance and bit movement.

6.2 Parallel Workloads for Evaluating Optimizations

This chapter discusses the compiler as a tool to lower parallel programs onto the EM^2 architecture, and single-threaded benchmarks are not adequate to demonstrate the behavior associated with parallel workloads: simple benchmarks have no inter-thread communication, no synchronization, and no migration. Section 1.2.2 describes the programs used to produce parallel workloads to evaluate compiler optimizations proposed in this chapter.

6.3 Optimizations for Reduced Instruction Count

As stated previously, reducing instruction count improves both core and migration efficiency, as well as bandwidth utilization in the system. We describe two optimizations for code size: `constant replication` and `commutative operators` to reduce scheduling overhead. We briefly discuss a relaxed consistency model optimization, but leave it for future work.

6.3.1 Commutative Operators

We discussed scheduling optimization in detail previously in Chapter 3. There are, however, other techniques we can employ to further reduce overhead incurred due to scheduling of variables onto the hardware stack in EM^2. One such technique is the reversal of commutative operators, the rewrite rules are listed in Table 6.2, and exemplified by a small DFG in Figure 6-3. The performance improvement is shown in Figure 6-4: instruction count decreases an average of 8%. Similar magnitude of improvement in stack access depth is also observed.

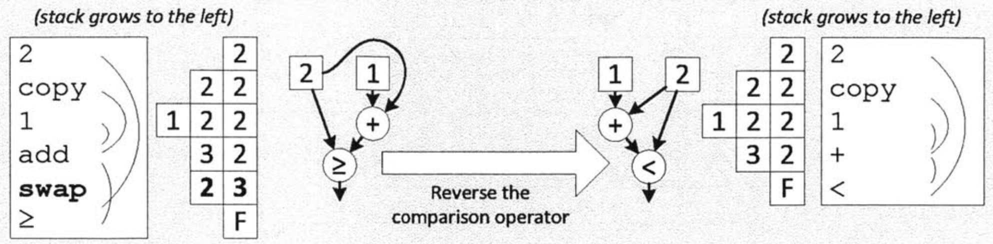

(stack grows to the left)
2
copy
1
add
swap
≥
Reverse the comparison operator
(stack grows to the left)
2
copy
1
+
<

Figure 6-3: Example of a DFG rewritten using the `commutative operators` optimization

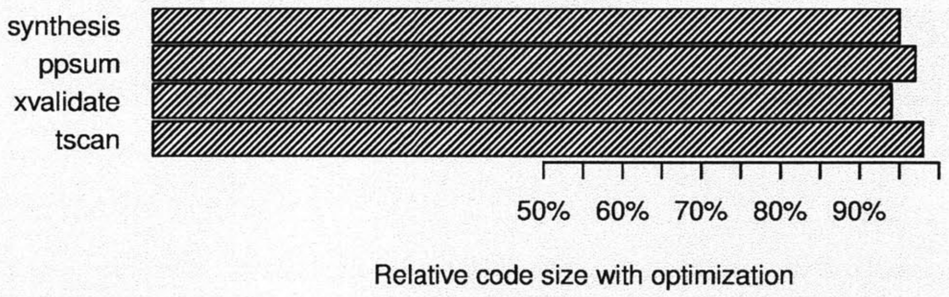


Figure 6-4: Instruction count after the `commutative operators` optimization

6.3.2 Constant Replication

As discussed in Section 2.2.6, each constant maps to a single expression that evaluates to the value of the constant. Each use of a constant is treated as a variable use, and is scheduled along with all other values on the stack, incurring scheduling overhead. An obvious optimization would instead insert the constant expression wherever it is needed, removing the constant variables from the scheduling problem entirely. Figure 6-5 shows an example DFG before and after the `constant replication` optimization. Figure 6-6 shows code size measured for several benchmarks after this optimization. As expected, the benefit of a simplified scheduling problem outweigh the cost of additional instructions needed to create the constant value. Most constants are 32-bits wide, and take 1.6 instructions to initialize. Without `constant`

Source pattern	Target pattern
add a b	add b a
and a b	and b a
or a b	or b a
xor a b	xor b a
mul a b	mul b a
comp_eq	comp_eq b a
com_ne	com_ne b a
comp_ugt	com_ule b a
comp_uge	com_ult b a
comp_ult	com_uge b a
comp_ule	com_ugt b a
comp_sgt	com_sle b a
comp_sge	com_slt b a
comp_slt	com_sge b a
comp_sle	com_sgt b a

Table 6.2: Rewrite rules for the `commutative operators` optimization

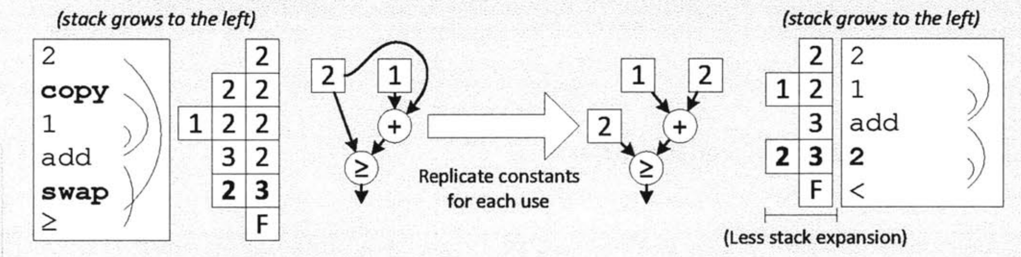

(stack grows to the left)
2
copy
1
add
swap
≥
Replicate constants for each use
(stack grows to the left)
2
1
add
2
<
(Less stack expansion)

Figure 6-5: Example of a DFG rewritten using the `constant replication` optimization

`replication`, at least one instruction is needed to copy the constant value for multiple uses. Scheduling overhead eliminated by `constant replication` clearly exceeds 0.6 instructions per constant because the overall code size is reduced by this optimization.

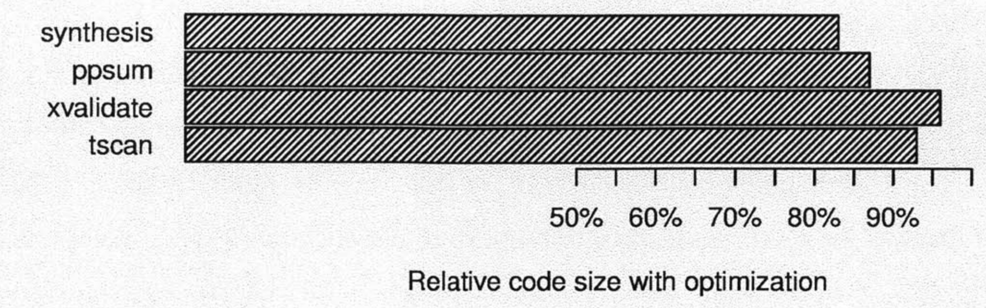


Figure 6-6: Instruction count after the `constant replication` optimization

6.3.3 Relaxed Memory Consistency

Another potential optimization comes in form of a relaxed memory consistency model: the compiler strictly maintains the ordering of all memory accesses for core consistency, which severely limits its ability to rearrange expressions. By relaxing this requirement, expressions can be ordered more freely, potentially allowing lower-overhead linearizations of the DFGs induced by each basic block. We do not implement or evaluate this optimization, leaving it for future work.

6.4 Optimizations for Core Misses

Core misses occur as a result of memory instructions addressing memory not associated with the current core. Although some of these instructions are a direct result of the programmer's data placement and access pattern, some correspond to variables in the stack frame - implicit state mapped to memory by the compiler. In this section, we discuss two optimizations: aggressive `static promotion` to remove these stack frame variables from memory, and promote them to variables in the hardware stack, and memory access clustering, which reduces core misses by grouping memory accesses.

6.4.1 Memory Access Clustering

To implement this optimization, we extend the cost function first discussed in Section 3.5.1 to favor DAG linearizations that place memory access instructions closely together. The idea behind this optimization is that accesses made within a basic block are likely to be to the same data structure, or closely related data structures, both likely affined to the same core. The memory access instructions are therefore likely to send requests to the same core, meaning a single migration may be able to cover all accesses without evictions, thereby reducing core misses. The scheme is not very effective due to EM^2's strong memory consistency, severely limiting the potential for memory access clustering. We do not present benchmark results for this optimizations, as they do not clearly illustrate the optimization as advantageous.

6.4.2 Static Promotion

A much more aggressive optimization is exemplified by `static promotion`. By treating all statically allocated memory (stack variables in C) as hardware stack variables not backed by memory. Doing so decreases the rate of memory accesses substantially, as shown in Figure 6-7, but increased scheduling overhead due to additional variables. Interestingly, this optimization works very well with partial context migration, as the hardware stack acts as a natural cache, sending values required for the instructions to be run at the guest core along with the context. Without `static promotion`, all accesses to these values would require memory loads, and would result in a core miss. Figure 6-8 shows the instruction overhead incurred by adding stack frame variables to the variables scheduled in the hardware stack. Figure 6-9 shows the reduction in migration rate, and specifically the core miss rate achieved by this optimization.

6.5 Underflow and Overflow Optimization

Underflow and overflow misses are unavoidable in EM^2 because the guest context does not back its hardware stack with memory. In other words, while the native core

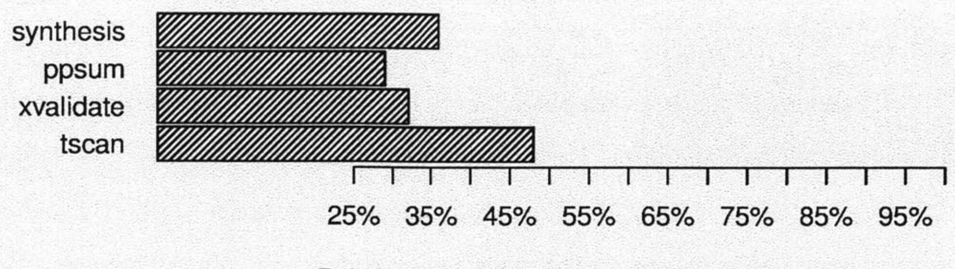


Figure 6-7: Memory access rate after the `static promotion` optimization

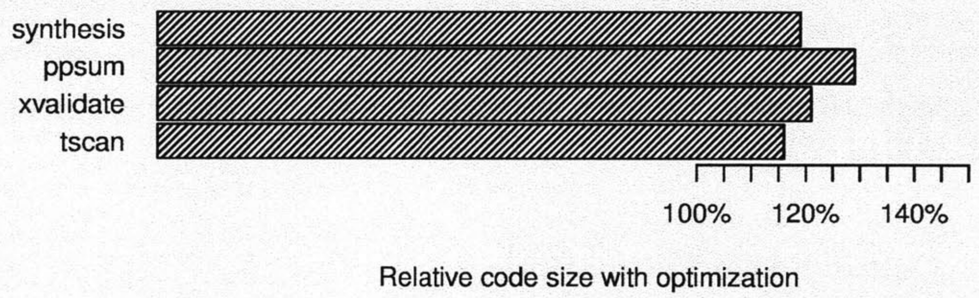


Figure 6-8: Code size after the `static promotion` optimization

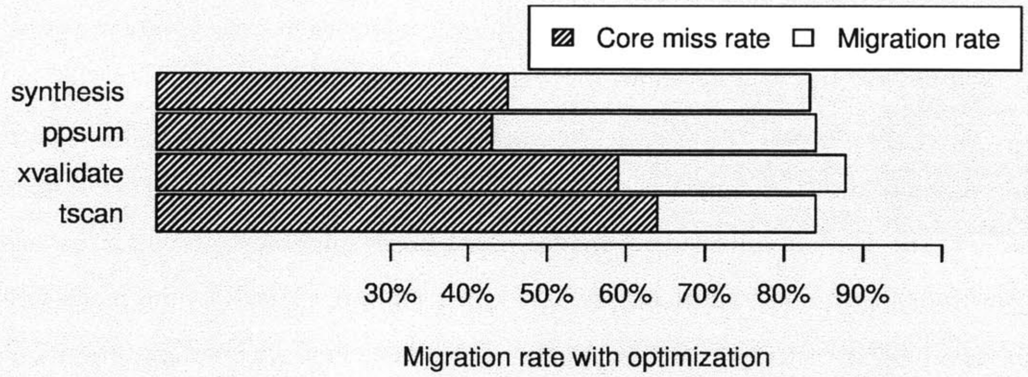


Figure 6-9: Migration rate after the `static promotion` optimization

is able to swap lower portions of the hardware stack in and out of memory, the guest context is limited to 16 entries, and is unable to store any more without "overflow-ing", causing an eviction back to the thread's home core, where the stack can be refilled from memory. Similarly, after migrating to a guest context, a thread may use all of the stack values made available with the migration and "underflow", again forcing an eviction back to the home core where subsequent stack entries are cached. These evictions are strictly overhead: they occur because resources implicitly needed to execute additional instructions on the guest context are unavailable (perhaps due to an inefficient instruction sequence). Even though we cannot completely eliminate these misses, we try to reduce them as much as possible, as they make up a significant portion of all migrations, and directly limit guest run length. In this section, we propose three optimizations to improve underflow and overflow behavior: `context size estimation` for partial context migrations, `expression reordering for improved run length`, and a library primitive to optimize bulk data transfers using migration and the hardware stack.

6.5.1 Context Size Estimation

Partial context migration can greatly improve run length: if the migration executes a reduction-like sequence, where the stack shrinks as values are reduced into a result, a larger migration generally corresponds to a longer run length, so we want to take more. Conversely, if we are reading or producing a lot of data at the guest core, we want to take less to avoid overflow, and return with a full stack.

Because the critical run length that makes a migration advantageous is rather short, and because run length is influenced by program phase and other dynamic variables, the EM^2 architecture provides a hardware migration predictor to select between migration and remote access based on a reinforcement learning mechanism. Although explicit selection of EM or RA is allowed, and we use this abundantly in the ABI library described in Chapter 5, we allow the predictor to learn whether to RA or to EM for compiled code. One thing we can do however is calculate the partial context size the predictor should take. Because each instruction has static producer and consumer behavior (as previously explored in Section 3.1), we can make intelligent decisions about context size. The `context size estimation` pass does

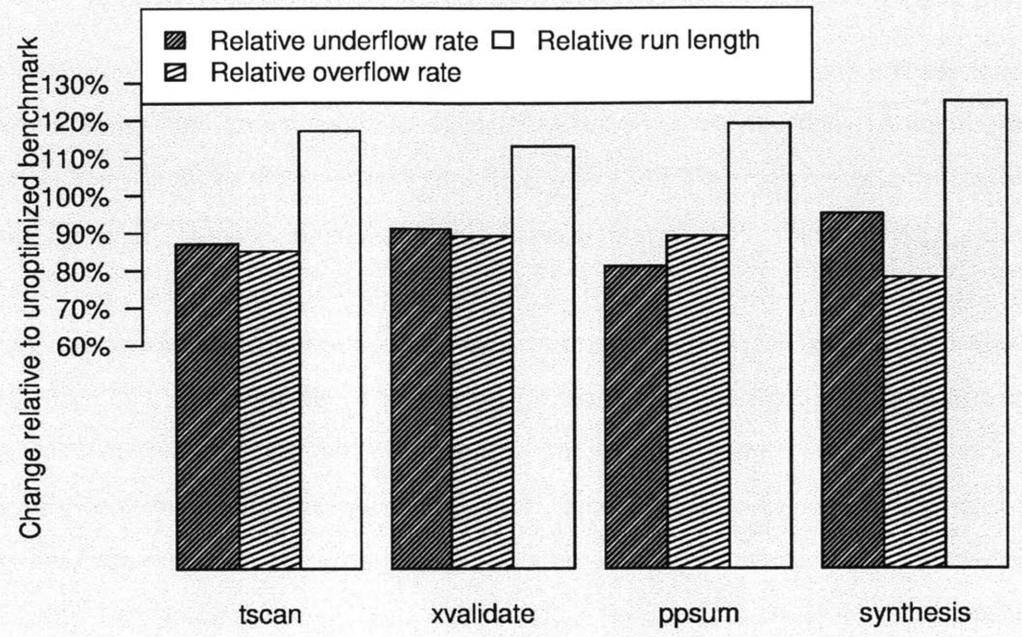


Figure 6-10: Relative rate of eviction due to under and overflow, and relative average run length after `context size estimation`

this by enumerating likely migration sites (memory accesses), and tracking maximum stack expansion and contraction after each potential migration. We try to maximize run length, choosing to migrate a well-sized context to maximize a utility function: we choose the largest N such that to execute N instructions remotely, we have to carry fewer than kN values. k may be tuned to improve results given relevant parallel benchmarks. We assume that all migrations are from home to guest for this optimization to simplify our model. To evaluate this optimization, we assume the predictor does not learn migration sizes, but instead reads them from metadata embedded in the instruction stream. We run the benchmarks on an ISA simulator and show the optimization effect on relative rates for overflow eviction, underflow eviction, and relative average run length in Figure 6-10. We set $k = 0.5$, although future work may benefit from tuning the coefficient on a per-application basis.

6.5.2 Expression Reordering for Improved Run Length

The `expression reordering for improved run length` optimization extends the above scheme by aggressively re-ordering basic block expressions. We modify the cost function used to select a DFG linearization in Section 3.5.1 to favor expression sequences with maximal expected run length after each memory access (likely migration sites). Although the optimization is applied at the basic block level, a similar approach can be used for trace optimization in the context of a JIT compiler [56] to produce efficient expression sequences optimized for loops and other hot paths of execution. Even in its current form, the optimization departs from efficient stack scheduling early in the compiler backend, instead favoring run length as the primary optimization goal. Because a sub-optimal stack schedule results in higher scheduling overhead, there is a trade-off between extended run length and compact code offered by this optimization. This optimization affects scheduling overhead in a significant way, so we defer its evaluation until Section 6.7, when we consider it in combination with other passes.

6.5.3 Bulk Data Transfer

A common pattern in multi-threaded workloads is bulk data transfer. EM^2 is designed to allow efficient read-write sharing via fine-grained execution migration and remote access, but read-only sharing introduces core contention and incurs a significant performance penalty. A local copy of read-only data is therefore useful for removing core communication for read-only data sets. Algorithm 13 shows a naive approach to bulk memory transfer: each iteration performs a remote load and a local store, meaning the predictor learns to use remote access, therefore performing N remote access loads to transfer N words of data, with $2N$ network messages of 2 flits each.

This operation can be implemented very efficiently as a library primitive by explicitly using partial context migration to transfer data on the hardware stack, instead

```
Input: $A_{remote}$ - address of an array in remote memory
$A_{local}$ - address of an array in local memory.
$N$ - size of the array to be copied.
Result: $Mem[A_{local} + i] \leftarrow Mem[A_{remote} + i] \forall i \in Z_N$.
begin
    for $i \in Z_N$ do
        Let $t = Mem[A_{remote} + i]$;
        $Mem[A_{local} + i] = t$;
```

Algorithm 13: Copy memory from remote memory via a naive compiled loop

of remote access. By unrolling the loop and performing several loads prior to storing the loaded values, we can use migration to transfer several values simultaneously. We can further optimize this algorithm by factoring out address computation, as shown in Algorithm 14 (simplified). As a result, only $\frac{N}{15}$ migrations are used, each requiring approximately 12 flits. Using an RTL implementation of the EM^2, we benchmark the bulk memory copy algorithm relative to the naive loop for several values of N, and show both completion time and bit movement improvements in Figure 6-11 and Figure 6-12, respectively. Although this is not strictly a compiler optimization pass, it may be possible for an aggressive compiler to discover the `memcpy` pattern in programs [47] in future work.

6.6 Last-Level Peephole Optimization

Most transformations described throughout this thesis rely only on local knowledge, and therefore may require cleanup to remove no-op instruction sequences. Typically, a compiler employs a last-level `peephole` optimization to remove redundant instructions and other inefficient sequences. These changes may be as simple as pruning jumps of unit length to somewhat complex transformations performing algebraic optimizations of arithmetic sequences. In the EM^2 compiler, we use a peephole optimization to perform a small set of localized optimizations:

```
Input: $A_{remote}$ - address of an array in remote memory
$A_{local}$ - address of an array in local memory.
$N$ - size of the array to be copied.
Result: $Mem[A_{local} + i] \leftarrow Mem[A_{remote} + i] \forall i \in Z_N$.
begin
    Let $M = \frac{N}{15}$;
    for $i \in (15 * Z_M)$ do
        // migrate to remote guest context on first remote load Let
        $t_0 = Mem[A_{remote} + i]$;
        Let $t_1 = Mem[A_{remote} + i + 1]$;
        Let $t_2 = Mem[A_{remote} + i + 2]$;
        Let $t_3 = Mem[A_{remote} + i + 3]$;
        ...
        Let $t_{14} = Mem[A_{remote} + i + 14]$;
        // migrate back to native core due to core miss on first store store
        $Mem[A_{remote} + i] = t_0$;
        $Mem[A_{remote} + i + 1] = t_1$;
        $Mem[A_{remote} + i + 2] = t_2$;
        $Mem[A_{remote} + i + 3] = t_3$;
        ...
        $Mem[A_{remote} + i + 14] = t_{14}$;
```

Algorithm 14: Copy memory from remote memory via an efficient bulk transfer

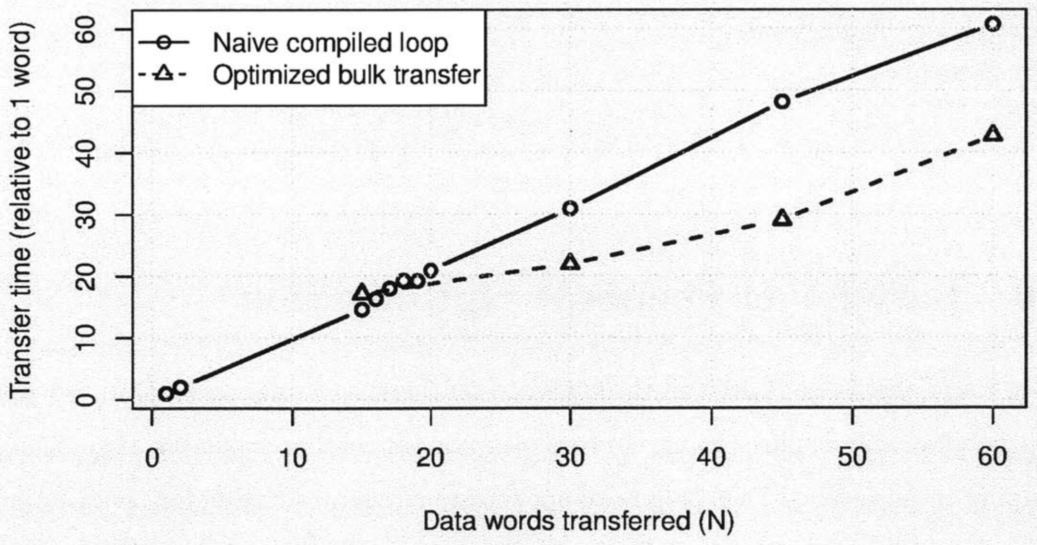


Figure 6-11: Relative completion time of an optimized bulk data transfer vs. a naive compiled loop

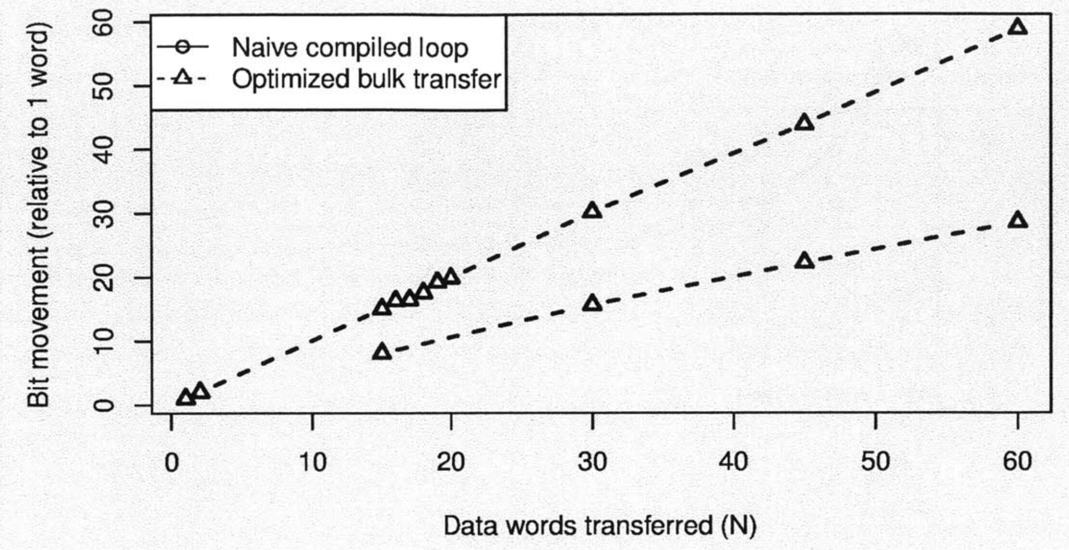


Figure 6-12: Relative bit movement of an optimized bulk data transfer vs. a naive compiled loop

- Offset memory addressing: A constant `add` producing an address for a memory access is folded into the memory instruction's offset immediate field, eliminating an addition instruction and a `push` instruction used to initialize the constant.
- Trivial control flow transfers: many basic blocks terminate with unconditional jumps transferring flow control to another basic block in the same function. Because the basic blocks are emitted by the back-end in reverse topological order, many of these jumps simply advance the `PC` by one. These instructions are removed.
- Deep stack accesses are remapped to the auxiliary stack, where possible.
- After expressions are flattened into a sequence of instructions, the `peephole` pass attempts to reorder instructions to remove stack manipulation.

The `peephole` pass does not process an entire program as a unit - doing so would be prohibitively expensive, and would not be useful because peephole optimizations can only be applied at the basic block level, where no control flow ambiguity exists. In

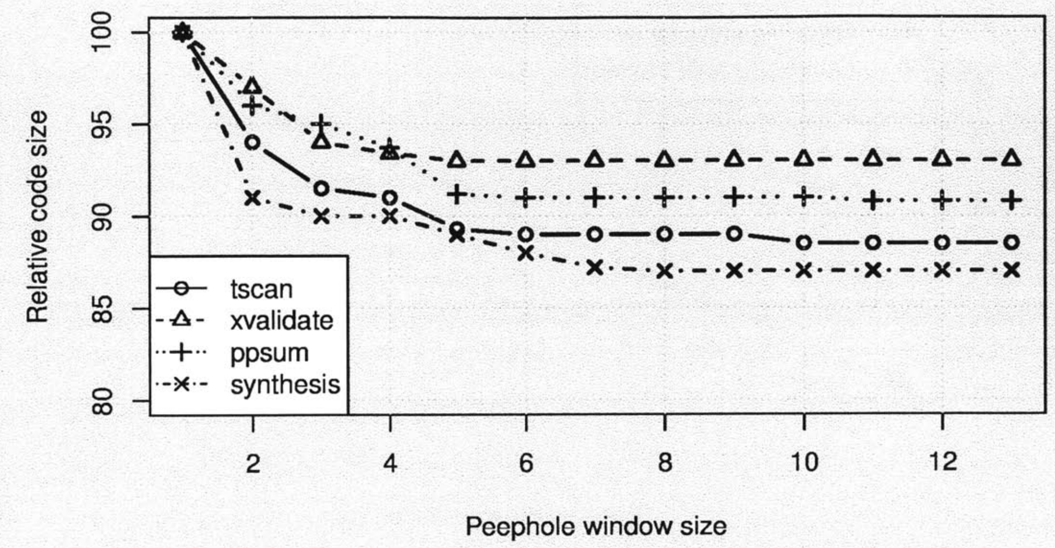


Figure 6-13: Effect of peephole window size on code size of result

fact, after the expression sequences of IR basic blocks are flattened into instructions, the number of basic blocks may increase because some expressions use branches and jumps internally to implement complex operators. After expressions are flattened, the average number of instructions per basic block is approximately 13 across the benchmarks evaluated in this thesis, meaning a `peephole` window in excess of 13 instructions is underutilized for many basic blocks. Moreover, increasing windows size yields diminishing returns with respect to code size, as illustrated in Figure 6-13. We fix the `peephole` window size at 6 instructions, as it yields good results with low compile-time overhead.

Any changes to the program made by the `peephole` pass may invalidate addresses and offsets used by the program (for example, if the program uses a `jump 9` instruction to advance to a particular basic block, and the previous block is shortened from 8 to 5 instructions, the control flow transfer must be updated to `jump 6`). For this reason, we enumerate all control flow transfers and their targets before applying the `peephole` optimization. The optimization may only be applied between transfers (over basic blocks), and the addresses of all labels (targets) must be re-evaluated afterwards.

6.7 Collective Evaluation of Optimizations

To evaluate the optimization passes, we must look at them as a unit, evaluating the cumulative effect of all optimizations on the program output. We use the optimized compiler output as a baseline to evaluate the effect of various subsets of optimizations described in this chapter. Some optimizations are clearly beneficial, and do not significantly interfere with other optimizations. Specifically, `constant replication`, `commutative operators`, and the `peephole` pass add little overhead, but produce superior instruction streams. We consider all three optimizations a unit, and do not consider the efficiency of subsets of these passes. `Static promotion` and `expression reordering for improved run length`, on the other hand are in conflict with other optimizations, meaning we must explore the trade-off offered by omitting or including these optimizations in the compiler flow. In this section, we benchmark the four parallel programs surveyed in this thesis, and show their code size (instruction count in the trace) in Figure 6-14, and migration rate in Figure 6-15, both normalized to the unoptimized baseline. To better illustrate the effect of the optimizations, we also show the relative rate of eviction due to underflow and overflow in Figure 6-16. Finally, we analyze the relative bit movement and benchmark completion time in Figure 6-17 and Figure 6-18, respectively.

`Static promotion` performs very well, reducing overall migration rates substantially by eliminating many core misses despite an increase in eviction rates due to additional variables scheduled onto the hardware stack. `Expression reordering for improved run length`, on the other hand has limited effect (likely due to constraints imposed by a conservative memory consistency model), and interferes with other optimizations, especially static promotion (likely because it schedules a large number of additional variables onto the hardware stack; sub-optimal schedules result in deep accesses, negating the goal of the `Expression reordering for improved run length` optimization). A relaxed memory consistency model would likely make many of the optimizations described in this chapter more effective.

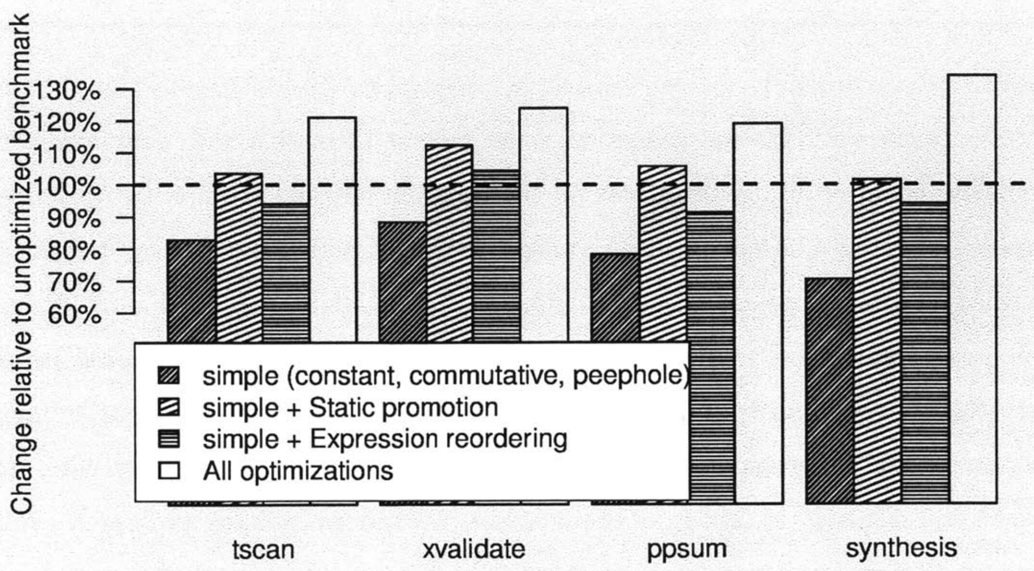


Figure 6-14: Effect of optimizations on benchmark code size, relative to unoptimized code

6.8 Summary

In this chapter we addressed a major limitation of the EM^2 compiler, namely its lack of awareness of key characteristics of the execution migration platform. Most workloads we expect to run on the EM^2 platform are highly parallel, and utilize migration for communication, so a compiler designed to optimize single-thread performance on a stack core is insufficient to produce high quality code for EM^2. To address this shortcoming, we designed and evaluated several compiler backend optimizations in an attempt to improve migration behavior and code size of compiled code. Although many opportunities for optimization remain unexplored, the passes described in this chapter significantly reduce frivolous migration by eliminating many opportunities for core misses, stack overflows, and stack underflows, resulting in lower overall eviction rate, and therefore better utilization of the EM^2 platform. Specifically, we describe three beneficial optimizations: expression rewriting to take advantage of commutativity of several EM^2 instructions, constant replication to trivialize scheduling of constant

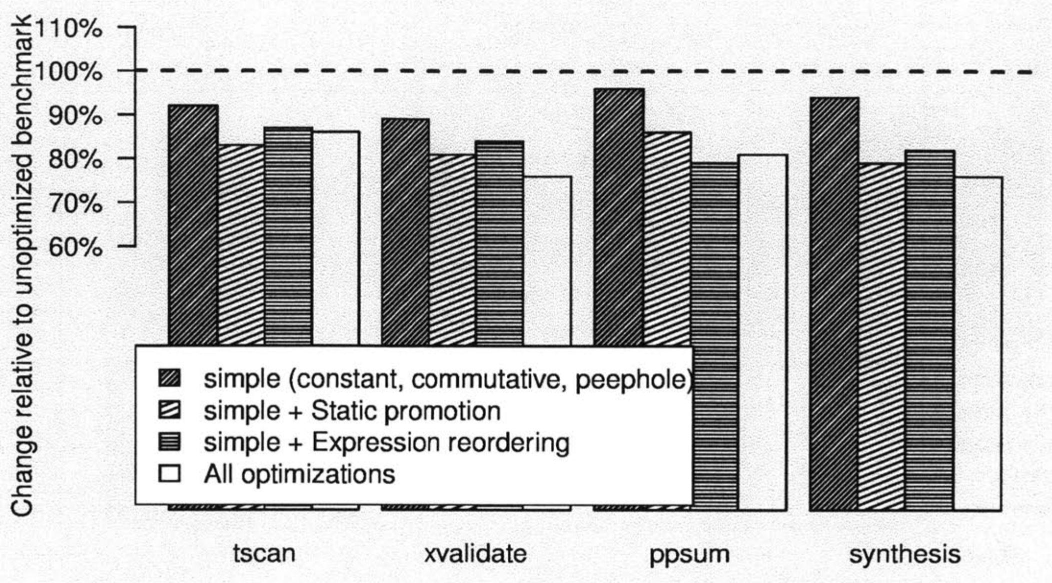


Figure 6-15: Effect of optimizations on benchmark migration rate, relative to unoptimized code

values and reducing total scheduling overhead and last-level peephole optimization to remove dead code and better utilize the auxiliary stack. We also describe two aggressive optimizations: removing statically allocated variables from memory in favor of the hardware stack (which breaks some unsafe code, but dramatically improves core miss rate), and reordering expressions to decrease the rate of evictions due to overflow and underflow. These are not always safe, but may improve performance and efficiency of compiled programs significantly.

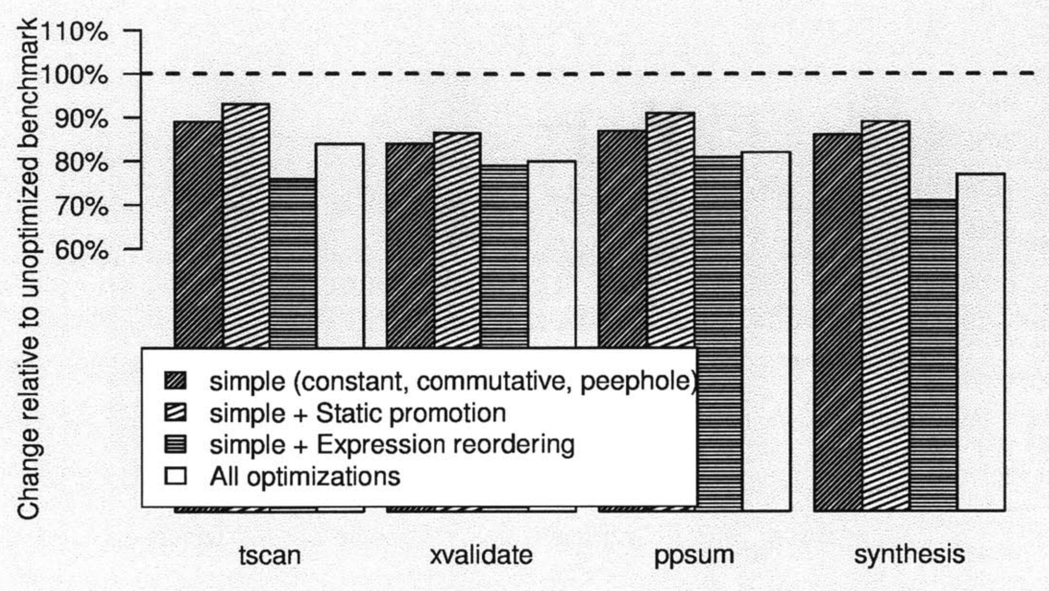


Figure 6-16: Effect of optimizations on benchmark underflow and overflow eviction rate, relative to unoptimized code

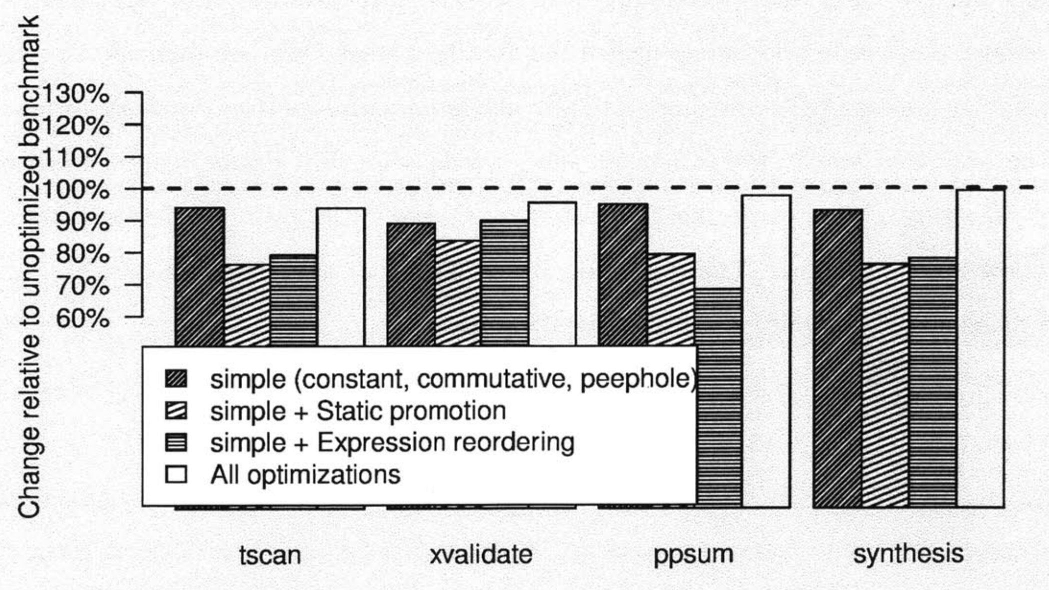


Figure 6-17: Effect of optimizations on overall benchmark bit movement, relative to unoptimized code

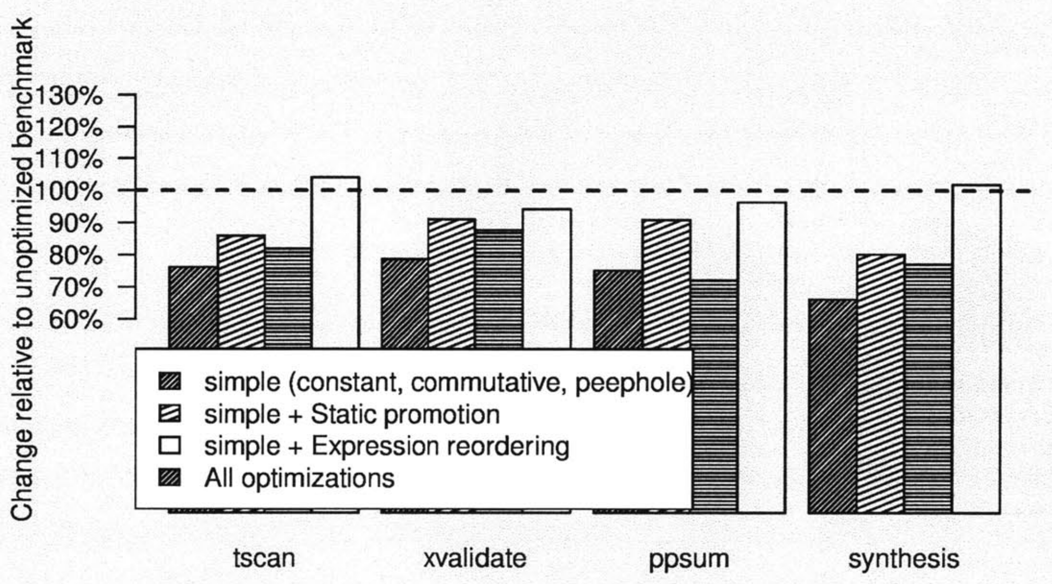


Figure 6-18: Effect of optimizations on overall benchmark completion time, relative to unoptimized code

Chapter 7

Conclusion

In this thesis, we discussed a productive programming and execution model for EM^2 and designed a compiler backend for LLVM to target the Execution Migration Machine. We also designed a library of optimized high-level primitives for thread communication and synchronization, as well as a set of compiler optimization passes to best reflect and utilize the underlying hardware by increasing migration run length and code density while reducing frivolous migration. We investigated costs associated with compilation to a stack-based architecture with hardware support for fine-grained migration in an effort to better understand ways in which a typical compiler flow may altered to yield better results when targeting EM^2. We use the insight gained to implement a series of algorithms to efficiently schedule program state onto the hardware stack, and produce several compiler optimizations to further improve compiler output.

7.1 Quality of Compiled Code

Before concluding the thesis, we would like to discuss the overall quality of code produced by the EM^2 compiler. Unfortunately, since no other compiler is available, we are unable to mount a credible comparative study and instead are forced to discuss the performance of the compiler in absolute terms. To do so, we compare the performance (in terms of completion time and bit movement) of several benchmarks compiled using the EM^2 compiler flow described in this work with a manual best-effort implementation of the same benchmarks. Of course the comparison is not quite "apples-to-apples" because the result depends largely on the quality of manual implementation: the compiler works at a higher level of abstraction, and is bound by a calling convention and other constraints, while the manual implementation may restructure computation in ways not available to a compiler because a programmer can restructure non-essential computation freely. Although a highly subjective approach, this comparison helps demonstrate the compiler is adequate. We compare the compiled and manual implementations of three benchmarks introduced in Section 1.2.2: `ppsum`, `xvalidate`, and `tscan`. Table scan relies on efficient migration to perform queries without excessive eviction. The cross-validation benchmark relies on a minimally-sized context and very long run length to migrate to several guests before returning to the native core. The parallel prefix sum benchmark relies on efficient bulk data transfer.

Completion time and bit movement measurements are shown in Figure 7-1 and Figure 7-2, respectively. Code size for compiled benchmarks is 150%-200% larger than the manually optimized assembly (executable size is significantly larger due to linked libraries, which may not be used by the benchmark). The parallel prefix sum benchmark compiles well because it derives most of its performance from an efficient bulk data transfer, usable via an optimized library primitive (described in Section 6.5.3. Compiled table scan also performs relatively well, as its overall performance does not dramatically deteriorate from a slight increase in code size and reduction in migration efficiency incurred during compilation. The cross-validation benchmark on the other hand performs poorly due to its reliance on a coarse-grained migration pattern: eviction has a strongly negative effect on this benchmark's performance. To better illustrate coarse-grain migration orchestrated by the manual implementation of `xvalidate`, we show the relative migration rates for compiled and manually optimized versions of this benchmark in Figure 7-3. The figure clearly shows that the manual implementation is able to avoid eviction and allow multiple guest contexts to be visited before returning the thread back to its home core. This is possible because the manual implementation explicitly orchestrates migrations to completely

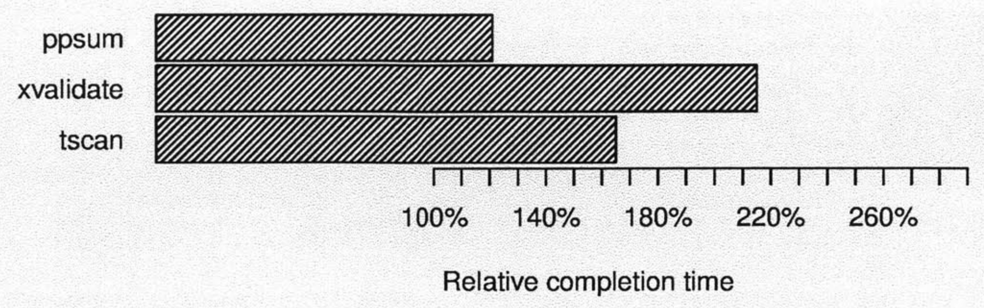


Figure 7-1: Performance of compiled parallel benchmarks relative to hand-optimized assembly

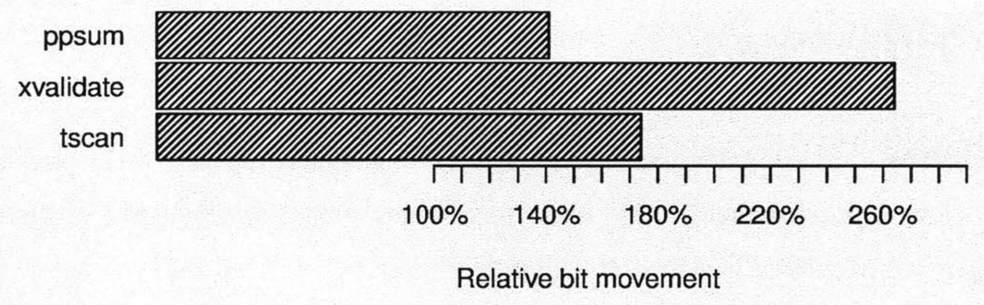


Figure 7-2: Bit movement in compiled parallel benchmarks relative to hand-optimized assembly

avoid unnecessary evictions and carries minimal data needed to process an entire data partition in each context.

One obvious shortcoming of the compiler is its inability to take advantage of coarse-grained migration orchestration due to lack of precise control over context size and run length. Doing so is possible (albeit difficult) in low-level assembly. Despite the somewhat high overhead of compiled code relative to full-custom assembly, the C programming language offers a very productive abstraction, allowing the programmer to quickly write large, complex programs, optimizing critical code primitives separately, where necessary.

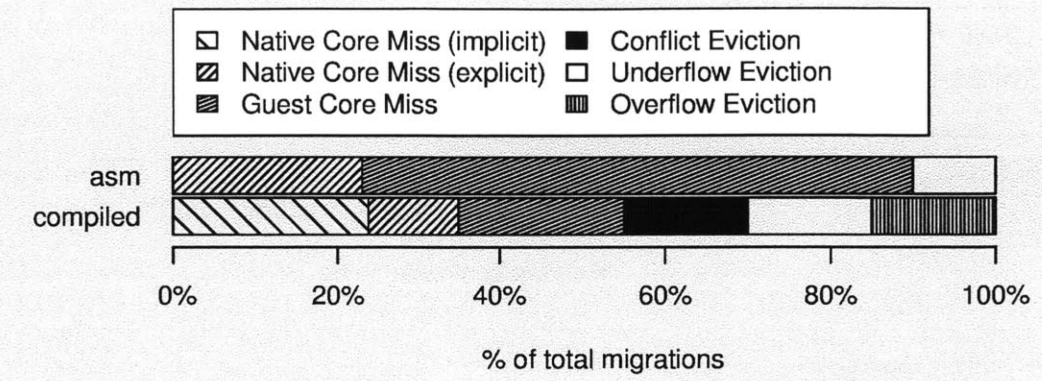


Figure 7-3: Migration behavior in compiled and hand-optimized `xvalidate` benchmark

7.2 Future work

Although the EM^2 compiler performs relatively well, much remains to be investigated before a programmer is able to write complex programs efficiently targeting the Execution Migration machine at a high level only.

7.2.1 Relaxed Memory Consistency Model

The EM^2 hardware supports a relaxed memory consistency model, but the compiler implements memory accesses in a very conservative way, ensuring full core consistency (making sure that any core sees the memory accesses made by any other core in the correct order). This conservative requirement limits the compiler's ability to reorder memory accesses, thereby limiting the ability of several compiler optimizations to improve compiler output. Many architectures expose an execution model with a much more relaxed memory consistency model, requiring the programmer to explicitly insert memory barriers to restrict memory access reordering for correctness. Exposing a relaxed memory model in the context of EM^2 may yield significantly more efficient code by allowing the compiler more freedom to reduce frivolous migration.

7.2.2 Memory Anti-Aliasing

Pointer analysis is notoriously difficult with C: the language treats a very large chunk of disorganized, type-free state (the memory) as a first-class citizen of the programming model. While this makes sense from the point of view of the underlying hardware, it makes compiler optimization very difficult wherever memory is accessed: data dependencies may be passed via memory in complex ways, meaning the compiler is seldom able to eliminate unnecessary memory operations, much less to reorder them. C code may be analyzed to some extent, determining which pointers are necessarily independent. A compiler may be able to use this information to better optimize program behavior. A more straightforward approach would be to compile from a higher-level language, one without a notion of a typeless memory, and therefore not requiring the solution to a complex inference problem to maintain correctness through optimization.

7.2.3 Profiler-Informed Optimization

Variable scheduling across basic blocks in the EM^2 compiler is driven by a simple heuristic (discussed in Section 3.6). By allowing the program to be evaluated, even with a randomized data set, we may be able to use the control flow statistics to better optimize the most frequently taken paths through the program. Doing so would dramatically reduce variable scheduling overhead due to control flow ambiguity (because the control flow ambiguity would be reduced by the profiler). The profile may also help disambiguate the memory access pattern, allowing significant optimizations by way of memory operation reordering.

7.2.4 Trace Optimization

Taking the idea of profile-based optimization further, we discuss the idea of just-in-time (JIT) compilation. By deferring last-level compilation and optimization until specific sections of the program are executed, the compiler would be able to iteratively optimize the program using dynamic knowledge such as the memory access pattern and control flow paths in the context of a specific data set the program runs on. Entire traces may be optimized using basic block scheduling techniques, resulting in lean, efficient code.

7.2.5 Inference of High-Level Operations

Many operators cannot be expressed in C unambiguously. For example, bulk memory transfer discussed in Section 6.5.3 can be implemented many different ways iteratively alone, but has a specific highly optimal implementation in the EM^2 platform. If the compiler is able to infer high-level operations such as movement of large blocks of data, synchronization, or use of specific data structures, it would be able to implement these high-level primitives in the most architecture-efficient manner available. Even if the programmer is aware of the eccentricities of the underlying machine, many low-level optimizations may not be expressible above the ABI, requiring the use of a library for efficient high-level operators.

7.2.6 Improved Data Placement

In the EM^2 machine, data placement has far-reaching implications on system behavior in the context of a multi-threaded program. Consider a workload where threads actively read and write a large data structure. The programmer may be able to distribute the structure across memory associated with all participating cores, spreading out the communication pattern. The programmer may also allocate the entire structure contiguously in memory associated with just one core, causing all accesses to be serialized at that core, resulting in high network congestion and a high eviction rate. Ideally, the compiler would be able to analyze a given data placement, either statically or using dynamic information from a profiler or a JIT engine, and detect performance bottlenecks, allowing the programmer to specify a better data placement. Taking the concept of intelligent data placement further, the compiler may be to statically place allocations to specific cores. For example, all data not shared with other threads is most appropriately allocated on the thread's native core to prevent unnecessary communication between cores. In a more advanced optimization, the compiler may cluster related data structures on adjacent cores (or the same core), or partition large data structures across multiple cores for better load balancing.

7.2.7 Dynamic data placement

Another approach to solving the data placement problem allows data to be moved in order to iteratively improve the data placement based on the observed access pattern. By employing a core address table (CAT), specific address ranges may be assigned to a new core, and this assignment may be updated dynamically to effectively move data, affecting migration behavior. By treating data structures as nodes with with weighted connections induced by the program's access pattern. We can treat this as a system of springs, and iteratively alter the data affinity to reduce total energy - produce a better data placement.

7.3 Summary

In this chapter, we conclude the discussion of the EM^2 execution model and compiler by comparing the compiler output to manual, hand-optimized implementations of equivalent functionality. We show that the compiler produces adequate machine code in absolute terms, and discuss its limitations. Finally, we discuss future directions in which this work may be improved.

Bibliography

[1] Mibench version 1.0. `http://www.eecs.umich.edu/mibench/`, August 2013. Last retrieved 2013-08-14.

[2] Writing an llvm backend. `http://llvm.org/docs/WritingAnLLVMBackend.html`, August 2013. Last retrieved 2013-08-14.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

[5] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on,* 1(1):6–16, 1990.

[6] Chris Bailey. Inter-Boundary Scheduling of Stack Operands: A preliminary Study. 2000.

[7] Chris Bailey, Huibin Shi, and Mark Shannon. Towards scalable parallelism with stack machines.

[8] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, May 2003.

[9] Jeffery A. Brown and Dean M. Tullsen. The shared-thread multiprocessor. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 73–82, New York, NY, USA, 2008. ACM.

[10] John Burkardt. C benchmarks. `http://people.sc.fsu.edu/˜jburkardt/c_src/c_src.html`, August 2013. Last retrieved 2013-08-14.

[11] David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[12] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize cmp cores on-the-fly. *SIGOPS Oper. Syst. Rev.*, 40(5):283–292, October 2006.

[13] Myong Hyon Cho, Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Deadlock-free fine-grained thread migration. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pages 33–40, New York, NY, USA, 2011. ACM.

[14] Intel Corporation. Intel-64 and ia-32 architectures software developer's manual.

[15] Intel Corporation, Carole DuLong, Mickey Gutman, and Mike Julier. *Complete Guide to Mmx Technology.* McGraw-Hill Professional, 1997.

[16] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[17] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using ssa-graphs. *SIGPLAN Not.*, 43(7):31–40, June 2008.

[18] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1033–, Washington, DC, USA, 1999. IEEE Computer Society.

[19] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, April 1989.

[21] Gary Granunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, June 1990.

[22] Mary H. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[23] John Hauser. Softfloat. `http://www.jhauser.us/arithmetic/SoftFloat.html`, August 2013. Last retrieved 2013-08-14.

[24] Douglas M. Hawkins, Subhash C. Basak, and Denise Mills. Assessing model fit by cross-validation. *Journal of Chemical Information and Computer Sciences*, 43(2):579–586, 2003.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[26] Damien Imbs and Michel Raynal. Trying to unify the ll/sc synchronization primitive and the notion of a timed register. In *Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, AINA '12, pages 326–330, Washington, DC, USA, 2012. IEEE Computer Society.

[27] Hai Jiang and V. Chaudhary. Compile/run-time support for thread migration. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 9 pp–, 2002.

[28] Gerry Kane and Joe Heinrich. *MIPS RISC architectures.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[29] Brian W. Kernighan. *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[30] Philip Koopman. Usenet nuggets.

[31] Philip Koopman. *Stack Computers: The New Wave.* Computers and their applications. Ellis Horwood Limited, 1989.

[32] Philip Koopman. A preliminary exploration of optimized stack code generation. *Journal of Forth Applications and Research*, 6, 1992.

[33] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[34] Chris Lattner and Vikram Adve. The llvm compiler framework and infrastructure tutorial. In *Proceedings of the 17th international conference on Languages and Compilers for High Performance Computing*, LCPC'04, pages 15–16, Berlin, Heidelberg, 2005. Springer-Verlag.

[35] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Christopher W. Fletcher, Michel Kinsy, Ilia Lebedev, Omer Khan, and Srinivas Devadas. Brief announcement: distributed shared memory based on computation migration. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 253–256, New York, NY, USA, 2011. ACM.

[36] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Ilia Lebedev, and Srinivas Devadas. The execution migration machine. Technical report, Massachusetts Institute of Technology, August 2013.

[37] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, March 2008.

[38] Martin Maierhofer and M.Anton Ertl. Local stack allocation. In Kai Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 189–203. Springer Berlin Heidelberg, 1998.

[39] Abid M. Malik, Tyrel Russell, Michael Chase, and Peter Beek. Learning heuristics for basic block instruction scheduling. *Journal of Heuristics*, 14(6):549–569, December 2008.

[40] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[41] P. Montuschi and L. Ciminiera. Algorithm and architectures for radix-4 division with over-redundant digit set and simple digit selection hardware. In *Signals, Systems and Computers, 1991. 1991 Conference Record of the Twenty-Fifth Asilomar Conference on*, pages 418–422 vol.1, 1991.

[42] The University of Illinois. Ncsa open source license (ncsa). `http://opensource.org/licenses/UoI-NCSA.php`, August 2013. Last retrieved 2013-08-14.

[43] Sriram Padmanabhan. *Data placement in shared-nothing parallel database systems.* PhD thesis, Ann Arbor, MI, USA, 1992. UMI Order No. GAX93-08416.

[44] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2007.

[45] Stephen Pelc and Chris Bailey. Ubiquitous forth objects. *EuroForth*, 2004.

[46] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. *SIGARCH Comput. Archit. News*, 37(3):302–313, June 2009.

[47] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. *SIGPLAN Not.*, 48(1):497–510, January 2013.

[48] David Seal. *ARM Architecture Reference Manual.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[49] Mark Shannon. A c compiler for stack machines. Master's thesis, University of York, United Kingdom, 2006.

[50] Mark Shannon and Chris Bailey. Global stack allocation (register allocation for stack machines). In *Procedings of EuroForth 2006*, 2006.

[51] Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Thread migration prediction for distributed shared caches. *Computer Architecture Letters*, Sep 2012.

[52] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[53] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient dag construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 93–102, New York, NY, USA, 1991. ACM.

[54] Inc. SPARC International. *The SPARC architecture manual: version 8.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[55] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3.* CreateSpace, Paramount, CA, 2009.

[56] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. *SIGPLAN Not.*, 36(11):180–195, October 2001.

[57] Kuo-Chung Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ICPP '94, pages 69–72, Washington, DC, USA, 1994. IEEE Computer Society.

[58] Guido van Rossum and Fred L. Drake. *Introduction to PYTHON 2.6.* CreateSpace, Paramount, CA, 2009.

[59] Li-Yi Wei. *Texture synthesis by fixed neighborhood searching.* PhD thesis, Stanford, CA, USA, 2002. AAI3038169.

[60] Jenq-Shyan Yang and Chung-Ta King. Designing tree-based barrier synchronization on 2d mesh networks. *IEEE Trans. Parallel Distrib. Syst.*, 9(6):526–534, June 1998.