Embedding Tree Computations
into the RAW Chip

by

Adrian Mihail Soviani

Submitted to the Department of Electrical Engineering
and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2002

© 2002 Adrian M. Soviani. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author: ........................................

Department of Electrical Engineering
and Computer Science
February 1, 2002

Certified by....

Anant Agarwal
Associate Director, MIT Laboratory for Computer Science
Thesis Supervisor

Accepted by .........

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Embedding Tree Computations into the RAW Chip

by

Adrian Mihail Soviani

Submitted to the Department of Electrical Engineering and Computer Science

February 1, 2002

in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

The main goal of this thesis is to provide coarse grain parallelism for the RAW chip. Coarse grain parallelism can be expressed by using instrumented function calls which are queued and executed on other processors. Function calls are reassigned to adjacent processors only. Subsequently, a computation tree is dynamically embedded into the processor mesh (Chapter 3).

A C library interface was developed to provide this mechanism: applications can spawn function calls, and wait at synchronization points (Chapter 4). Two implementations of the spawn interface were built as part of the thesis project: i) a system that runs on the RAW chip using interrupting messages to implement work scheduling and synchronization (Chapter 5); ii) a generic spawn library based on POSIX threads that uses shared memory and locking (Chapter 6).

Embedding computation trees into 2D processor meshes proved to be successful for a selection of tree structures and mesh sizes. Two models were used for testing: a perfect binary tree, and a Fibonacci tree.

On RAW both benchmarks produced similar speedups: $3.5 - 4$ for $2 \times 2$, $7 - 8$ for $3 \times 3$ when parallelism exceeds $2^7 - 2^9$; $12 - 13$ for $4 \times 4$ when $P \geq 2^9 - 2^{10}$. These numbers varied as a function of mesh size, problem size, work granularity, and system settings - spawn stack size, work scheduler policy. System overhead becomes negligible once granularity is sufficiently large.

On the other hand, simulations using the POSIX system showed speedups of $35 - 50$ for $8 \times 8$, $20 - 30$ for $6 \times 6$, and $95 - 110$ for $12 \times 12$. Empirically, we can conclude that the spawn framework is efficient for large processor meshes.

Thesis Supervisor: Anant Agarwal
Title: Associate Director, MIT Laboratory for Computer Science
Acknowledgments

I would like to thank Prof. Anant Agarwal for his support and guidance throughout all stages of my research. I would also like to thank Walter Lee for his advice and long discussions that provided me invaluable feedback on the project. Finally I would like to thank the people from the Computer Architecture Group who developed the tools needed to build and benchmark the system.
Contents

1 Introduction ........................................... 15
   1.1 Thesis Organization .............................. 16

2 The RAW Project ...................................... 19
   2.1 Design Overview ................................ 19
       2.1.1 Static and Dynamic Networks .............. 19
       2.1.2 Deadlocks .................................. 20
       2.1.3 I/O Ports .................................. 21
       2.1.4 Memory .................................... 22
   2.2 Instruction Level Parallelism .................... 22
       2.2.1 Benchmarks ................................. 23

3 Coarse Grain Parallelism ............................ 25
   3.1 C Dialects ....................................... 25
   3.2 Computation Structures .......................... 26
       3.2.1 Search Algorithms ......................... 26
   3.3 Work Scheduler ................................ 27
       3.3.1 Master-Slave Model ....................... 28
       3.3.2 Randomized Work Stealing ............... 28
       3.3.3 Randomized Tree Embedding .............. 28
   3.4 Design Alternatives ............................. 29
       3.4.1 Load Balancing ............................ 29
       3.4.2 Synchronization ............................ 30
Conclusion

7.1 Future Work

A RAW Spawn Interface

B RAW Spawn Examples

C RAW Spawn Source Code Extracts

D RAW Spawn Outputs

E Pthreads Spawn Interface

F Pthreads Spawn Examples

G Pthreads Spawn Outputs
List of Figures

2-1 The RAW processor is a 2D mesh of tiles, each consisting of a main processor and a programmable switch 20

4-1 Definition of a "spawnable" function: fib 32
4-2 Processor state history: traversing a 1023 size heap on 4 x 4 33
4-3 Heap traversal: work scheduling on 2 x 2 tiles 36

5-1 The 3 phases of a RAW tile: stealing, working, and reporting 41
List of Tables

2.1 Speedup benchmark: rawcc vs. Machsuif MIPS compiler 23

5.1 RAW spawn message types 40
5.2 Average leaf work - Bin 46
5.3 Average leaf work - Fib 46
5.4 Overhead of spawn function calls - Bin 47
5.5 Overhead due to steal requests - Bin 47
5.6 Constant cost overheads for $P = 2^3$ - Bin 47
5.7 Increasing parallelism for constant granularity - Bin 49
5.8 Tradeoff between granularity and parallelism - Bin 49
5.9 Tradeoff between granularity and parallelism - Fib 49
5.10 Increasing spawn stack size - Fib 50
5.11 Increasing parallelism and stack size for constant granularity - Fib 50

6.1 Speedup for binary heap traversal 56
6.2 Speedup for Fibonacci tree traversal 57
Chapter 1

Introduction

The RAW architecture represents a simple, highly parallel VLSI architecture that exposes its low level resources to the software layer. Only a minimal set of mechanisms are implemented in hardware, greatly decreasing the complexity of the hardware layer. The software layer - a compiler for instance - has a great flexibility in programming the parallel chip, possibly allowing for an optimal resource allocation for a specific task [1].

The RAW multi-tile chip is somewhat similar to FPGAs; both architectures use a 2D mesh of programmable cells which are interconnected by fast communication logic. However, FPGAs have a finer cell granularity, do not include a large distributed memory, and do not support instruction sequencing.

RAW machines are also similar to Very Long Instruction Word processors. Both architectures have a large register name space, a distributed register file and multiple memory ports [7]. Moreover, RAW machines have individual instruction streams for each tile, providing a greater flexibility. Mapping an algorithm to one of these architectures is based on exploiting Instruction Level Parallelism by partitioning, placing, routing and scheduling the code on the 2D tile mesh. Rawcc is an effective compiler based on this technique [2].

A different approach to exploiting parallelism on the RAW architecture is relying on coarse grain parallelism. This thesis employs this approach. Some disadvantages of ILP can be overcome by explicitly partitioning the code and data structures at a
higher level. The difficulty of this method consists of finding a good way to expose coarse grain parallelism.

Solving this problem has been a persisting research issue, and significant progress has been made. Existing solutions for network connected clusters and shared memory machines involve designing a language extension and/or a communication library - Implicitly Parallel C, Data Parallel C, mpC programming language, MPI, Cilk. A desirable goal is to model the performance of specific algorithms for a given machine topology and software layer. Unfortunately, applications need to be modified to use the above interfaces. Moreover, some systems are not well suited to all classes of algorithms.

1.1 Thesis Organization

The research supporting my thesis will address a less general problem: executing computation trees on the RAW chip. Allowing little slack regarding shared memory, most search problems fall into this category. By narrowing down machine topology and computation structure we can build a better system than in the general case.

The thesis project attempts to reach this goal by developing the Spawn system. Spawn exploits coarse grain parallelism inherent to recursive computations: function calls can be executed concurrently on different processors. Spawn takes advantage of RAW topology by embedding computation trees into the 2D mesh: function calls are reassigned to adjacent processors only, paying constant parent-child communication cost for any mesh size. In order to use Spawn, applications need to add spawn function calls and synchronization points. These primitives add a significant overhead, which becomes negligible once work granularity is sufficiently large. Shared memory is not supported by the system, but it can be used if applications enforce their own policies suitable to specific hardware. Load balancing might not be optimal due to work assignment restrictions.

This thesis is organized in the following main sections: Chapter 2 presents the RAW chip and its ILP compiler. Exploiting coarse grain parallelism on 2D processor
meshes is discussed in Chapter 3. The Spawn system is described in Chapter 4. Chapter 5 outlines the RAW implementation of Spawn, and it analyzes experimental results. The pthreads implementation of Spawn and simulation results are presented in Chapter 6. Finally, Chapter 7 concludes this thesis, also discussing follow-on work.
Chapter 2

The RAW Project

2.1 Design Overview

This section outlines the design specification of RAW, the architecture on which the Spawn system is implemented. The RAW chip is a 2D mesh of replicated tiles. Each tile is connected point-to-point to its neighbors via two static and two dynamic networks. A tile consists of a main MIPS-like processor, a programmable static switch, and a dynamic router. Main processors have data and instruction memories; switches have only instruction memory (Fig. 2-1, [1]).

Main processors use an augmented MIPS 2000 instruction set - special registers access switches or hold configuration settings. Static switches use a subset of the MIPS instruction set, supporting only routes and control-flow operations. Each individual main processor and static switch is programmed separately - a $n \times n$ RAW machine requires $2n^2$ programs. Messages passing on the switch don’t interrupt the main processor. However, it is possible to setup interrupt handlers that are triggered by messages received from the dynamic network.

2.1.1 Static and Dynamic Networks

RAW provides two static networks - cst, cst2, and two dynamic networks - cgn, cmn. cmn is typically reserved for memory access, while the other networks are available
Figure 2-1: The RAW processor is a 2D mesh of tiles, each consisting of a main processor and a programmable switch

to the user.

Messages sent via static network are routed according to switch programs that are typically generated at compile time. Each tile processor knows the order and destination of incoming and outgoing data words; no headers are used by static messages. The static scheduling of messages guarantees infrequent stalls if the switches are loaded with correct code. However, dynamic messages might delay execution of static transfers, skewing synchronization of static message scheduling [6].

Messages sent via dynamic network start with a header. The header contains destination tile address, funny bits (direction of output port for edge tiles), message length (up to 31 words), 4 user bits, and source tile address. The header is used for routing, and it is dropped at destination unless it exits one of the I/O ports. Globally, routing is based on coordinate order. The message is forwarded from one dynamic router to the next one, until it reaches its destination. No more than a single tile is traversed every clock cycle.

2.1.2 Deadlocks

Reading from the switch ports is blocking, as well as writing when buffers are full. Circular dependencies between tiles can result in deadlock if each processor blocks either i) reading from the next processor in the chain while no data is available; or ii) writing to the next processor while buffers are full. For the static network it is the
responsibility of the compiler to avoid static scheduling deadlocks. For the dynamic network there are a few protocols that guarantee deadlock-free communication:

1. Making sure all messages are sinkable. Every processor has to be able to fully read all received messages without blocking. Using an interrupt handler, incoming messages can be dispatched by storing values, or by copying the entire message to local memory.

2. Making sure reads and writes don’t block indefinitely. Such an example is a client-server protocol where servers and clients are disjoint sets; client-server and server-client links are disjoint, too. All clients have only one outstanding message. After sending a request clients wait for the answer, therefore server writes will not block indefinitely. Request messages are not blocked by reply messages, therefore server reads can succeed. One example is partitioning tiles in North (client) and South (server) tiles. The memory network follows a similar protocol.

2.1.3 I/O Ports

The 2D processor mesh has a boundary that corresponds to the pins of the chip package.

These pins are connected to some of the static and dynamic network links that connect the tiles. Depending on package type, all network links are connected to pins, or only a few of them can be connected. In the latter case the compiler will take into consideration the configuration of the chip, and it will not use the network for the pins that don’t exist.

The I/O design has a great flexibility and scalability; in the worst case the communication limitation is $2n$ for a $n \times n$ tile processor. The compiler has direct access to the I/O interface, an application being able to use the full bandwidth of the pins.
2.1.4 Memory

Each main processor has 32 Kb SRAM of data memory, and 32 Kb of instruction memory. The RAW chip typically has DRAM memory banks connected to the East and South I/O ports. The RAW tiles access the memory banks to map large memory spaces from external memory to the small local memory/cache. Reading and writing memory pages is performed using the memory dynamic network, following the client-server deadlock protocol. This imposes some restrictions on what memory banks can be used by each tile.

In software caching mode each tile processor uses the memory network to virtualize data memory; any access to an invalid address (greater than 32K) returns invalid data. In hardware caching mode the RAW chip implements a 2-way set associative cache. Cache misses freeze the tile processor until data is available [6]. Instruction memory is always used in software caching mode.

2.2 Instruction Level Parallelism

Both coarse and fine grain parallelism can be exploited on the RAW chip. If Spawn relies on coarse grain parallelism, the Rawcc compiler uses fine grain parallelism. This section describes how instruction level parallelism can be exploited on RAW.

The RAW architecture is related to the VLIP machines; both architectures rely on statically scheduling ILP. If VLIPs have a single flow of control, the RAW machine allows a different flow of control for each tile. Moreover, the RAW machine provides a software exposed scalable interconnect between the tiles [2].

Instruction level parallelism is exploited on the RAW chip by partitioning a sequential C / Fortran program into sets of instructions that can be executed concurrently. There are as many code partitions as tiles on the chip. Analyzing data dependencies, communication cost between partitions is minimized. This operation corresponds to the following stages of Rawcc: “Initial Code Transformation”, “Instruction Partitioner” [2]. These partitions are generated using a greedy clustering algorithm.
The next step is “Global Data Partitioner”. Data elements that need to be shared among basic blocks are clustered in blocks that will be mapped to tiles. The goal is to preserve data locality: data that is accessed more frequently by a thread is clustered together. The task of partitioning data is distinct from the task of placing the partitions.

The previous steps assumed all code and data partitions are connected by a full interconnect; unfortunately this assumption doesn’t hold in practice. Having the data and code partitions, the “Data and Instruction Placer” maps one-to-one virtual data/code tiles to physical tiles. The cost of communication corresponds to the physical properties of the network. A greedy algorithm is used to minimize communication cost: two tiles are swapped if this operation reduces the overall cost.

The last step represents routing the messages on the 2D network and computing a global schedule that minimizes the overall runtime: “Communication Code Generator” and “Event Scheduler”. Routing is done using dimension-ordered routing; this ensures that any scheduling that doesn’t deadlock for an event sequence doesn’t deadlock on a scheduling that maintains the same order of events [2]. This is particularly useful if we consider dynamic network events that add delays, such as cache misses.

2.2.1 Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N=1</th>
<th>N=2</th>
<th>N=4</th>
<th>N=8</th>
<th>N=16</th>
</tr>
</thead>
<tbody>
<tr>
<td>fppp-kernel</td>
<td>0.48</td>
<td>0.68</td>
<td>1.36</td>
<td>3.01</td>
<td>6.02</td>
</tr>
<tr>
<td>btrix</td>
<td>0.83</td>
<td>1.48</td>
<td>2.61</td>
<td>4.40</td>
<td>8.58</td>
</tr>
<tr>
<td>mxm</td>
<td>0.94</td>
<td>1.97</td>
<td>3.60</td>
<td>6.64</td>
<td>12.20</td>
</tr>
<tr>
<td>life</td>
<td>0.94</td>
<td>1.71</td>
<td>3.00</td>
<td>6.64</td>
<td>12.66</td>
</tr>
<tr>
<td>jacobi</td>
<td>0.89</td>
<td>1.70</td>
<td>3.39</td>
<td>6.89</td>
<td>13.95</td>
</tr>
</tbody>
</table>

Table 2.1: Speedup benchmark: Rawcc vs. Machsuif MIPS compiler

Using the Rawcc compiler proved to be successful for many real applications. The results presented in Table 2.1 are taken from “Memory Bank Disambiguation using
Modulo Unrolling” [1]. Looking at Figure 2.1, we notice speedup is super-linear for “jacobi”, it is almost linear for “life” and matrix multiplication, and it decreases for “fppp-kernel”. This suggests ILP speedup efficiency varies significantly for each application type. If the communication pattern is relatively simple for “life” - each node interacts with its closest neighbors, “fppp-kernel” data dependencies are not obvious. This added complexity can account for the decrease in speedup.
Chapter 3

Coarse Grain Parallelism

Exploiting coarse grain parallelism is an alternative to fine grain parallelism. Typically this is done by defining an extension to a familiar language, or by providing a communication library. This chapter first outlines a few commonly used parallel systems. Second, it is shown that search algorithms perform well on Spawn, as opposed to other parallel systems. Third, alternative dynamic work schedulers are analyzed. Finally, a discussion on design alternatives for Spawn ends this chapter.

3.1 C Dialects

There are many C dialects that provide parallelism on multi-processor machines; these languages use specialized compilers. Spawn and MPI provide a library interface instead. Computation can be mapped statically or dynamically to processors.

1. Data Parallel C defines new data types that are shared across all processors, while the control flow is unique. This language maps well matrix operations, and more generally scientific computations.

2. Implicitly Parallel C implements a mechanism of locking shared resources and synchronization based on data structures semantics. IPC is designed for shared memory systems.

3. The Message Passing Interface offers C primitives for building a network of
nodes, and exchanging messages between nodes according to a specified pattern. Each communication activity is typically performed by all nodes at the same time, followed by global synchronization.

4. The Cilk programming language adds “spawn” and “sync” points to the control flow of a program. At spawn points idle processors can be assigned a subproblem, while sync points require all subproblems to be solved before continuing execution. Cilk is currently supported for shared memory systems; distributed Cilk was supported earlier. See [5] for a detailed manual on the Cilk language. Semantically the Spawn interface resembles a subset of the Cilk C extension.

3.2 Computation Structures

The Spawn framework doesn’t perform the best for all problems. This section shows why search algorithms are better suited to Spawn, as opposed to other parallel systems.

Computation structures can be analyzed in terms of data-flow or control-flow. A typical scientific computing problem like Jacobi or matrix multiplication have a simple data-flow. Such computations can easily be mapped onto $n \times n$ processor meshes by mapping equal slices of data to tiles, and preserving locality of data access. The communication pattern is predictable, and the amount of work is equal for all tiles. This type of coarse grain parallelism can be expressed nicely in Data Parallel C, or using a messaging library like MPI.

There are problems that have an unpredictable data-flow, but which can be described easier using the control-flow paradigm; such an example is searching for the best move in a chess game, or finding the maximum flow in a graph.

3.2.1 Search Algorithms

This section presents some of the most common search algorithms, showing why a dynamic work scheduler is required to exploit coarse grain parallelism.
Search algorithms explore a large search space, trying to find an optimal solution. Searching consists of generating a tree of possible events. A priori knowledge about the problem can be used to prune the search tree. Many times the optimal solution is computationally hard to find. Alternatively, approximations of the optimal solution are found using heuristics or approximation algorithms.

A few standard search algorithms are: i) alpha-beta cut exploration of min-max game trees; ii) A* - "branch and bound" a search tree, using a pessimistic heuristic estimator for the distance to the solution; iii) IDA* - iterative deepening A* having the advantage of not storing the search boundary; and iv) back-tracking - the most general algorithm for traversing a search tree.

Parallelizing such algorithms can be done by concurrently computing the outcome of each subtree, waiting for subtree computation to finish, then merging the results. However, data dependencies between branches can stall computation. Also, the performance of pruning the search tree might depend on the order in which the tree is evaluated. But most importantly, the structure of a search tree is not known before the tree is actually generated. Consequently, it is hard to evaluate the work load associated with each subtree, and statically map work to processors.

The Spawn system is well suited to this class of problems since its work scheduler is dynamic. Furthermore, the search tree is embedded into the 2D processor mesh, achieving constant cost parent-child communication. The parallel languages presented in Section 3.1 don’t allow dynamic work scheduling or don’t preserve locality of parent-child communication.

### 3.3 Work Scheduler

Due to the indeterministic\(^1\) nature of most search algorithms, a dynamic work scheduler is required to balance work among processors. We will study how alternative schedulers map work on the RAW chip, underlining the advantages of randomized tree embedding that is used by Spawn.

---

\(^1\)i.e. it is hard to approximate outcome given the input
3.3.1 Master-Slave Model

The master-slave model is a standard all purpose solution for dynamically distributing work to a number of processors. For the RAW chip, one tile could act as a master assigning tasks to the other slave tiles.

However, each time a task is assigned or results are reported the master and the slave tiles exchange data. The 2D mesh used by RAW provides a high communication bandwidth between any two tiles. Still, bandwidth can become a bottleneck for the links close to the master tile, especially when the number of tiles is large. Also, such a mapping algorithm doesn’t use in any way the properties of the 2D interconnect: any 2 tiles are seen identical.

3.3.2 Randomized Work Stealing

Each idle tile tries to steal a task from a randomly selected tile. If the attempt is not successful, another tile is selected, and so on. This strategy is used by Cilk for its SMP implementation. Load balancing is very good, since any idle tile can steal any available task. However, for a large number of processors this strategy can result in significant communication overhead - $O(n^2)$. Neither in this case it is taken advantage of the 2D topology of the mesh.

3.3.3 Randomized Tree Embedding

Any computation expressed as a parallel tree traversal can be embedded into a 2D mesh of processors paying little parent-child communication cost; for dilation 1 communication is minimized while load balancing might be an issue.

The Spawn project uses this framework, and it allows only neighbor tiles to steal work. Load balancing is achieved by random stealing, as opposed to statically embedding a tree in the mesh. Therefore, a tree is dynamically generated by randomly adding leaf tiles to the tree of already selected tiles.

Randomized tree embedding is more tolerant to some worse case scenarios. Imagine a computation tree is drastically unbalanced, and $n$-th level node $X$ is assigned
most work. The static tree embedding will probably keep idle all tiles not in \( X \)'s subtree. On the other hand, randomly generated trees will keep idle only the \( n \) tiles connecting root to \( X \). See Chapter 5 and Chapter 6 for experimental results of embedding Fibonacci and binary trees.

Computing Fibonacci recursively doesn’t require significant memory overhead. On the other hand, a typical search problem relies on shared memory to keep its global state. A machine that consist of a large 2D array of processors with large local memories can still achieve good results. The entire state can be copied from parent to child while no shared memory is used. Assuming state data has constant size or is proportional to problem size, inter-tile communication has a constant cost for any mesh size. Subsequently, memory access doesn’t become a scalability bottleneck.

### 3.4 Design Alternatives

#### 3.4.1 Load Balancing

The tree embedding scheduler doesn’t guarantee each tile has work to do. We only know for sure the root tile runs until the entire problem is solved; the adjacent tiles have a good chance of getting work from root, unless the tree is significantly unbalanced.

A possible improvement is requiring “center” tiles not to spawn too small a task. This would increase chances for tiles far from center to receive tasks. Unfortunately, it is hard to approximate the size of a task.

Increasing the reach of each tile by redefining the notion of “closest neighbors” can be helpful. By considering neighbors all tiles at \( L_1 \) distance 2, the depth of the tile tree is reduced by a factor of 2; each node has 12 neighbors. Increasing this parameter trades bandwidth for connectivity [4].

3.4.2 Synchronization

After finishing, a tile waits for its children to finish. The wait state is undesirable, under-utilizing resources. In the worse case only “leaf tiles” are doing work, while all others are waiting to aggregate data. If work granularity and problem size are well chosen, this scenario can be avoided (Section 5).

The synchronization step is not computationally intensive, allowing a tile to run other tasks, too. Having tiles run multiple tasks avoids the wait state problem. However, only one task can be active, all others waiting to aggregate data. This solution can lead to a blowup in memory access, since local memory will be insufficient for the needs of all tasks.

The waiting problem can be avoided by delegating the aggregation operation to a child who still works on its job. The ownership of the parent is transferred to the child, and all tiles involved in the computation need to be informed. Unfortunately, this algorithm doesn’t preserve locality of data transfers.

3.4.3 Memory Footprint, Locality

Each tile has 32Kb local memory; this is probably insufficient to store all local data. The simpler solution is using local memory as caches backed up by the external memory. Each time a task is assigned or reported, the communication between tiles translates to a copy between caches and memories. Such a solution has various limitations, not scaling with the number of tiles [3].

A possible improvement is managing data memory explicitly and faulting pages from neighbor tiles as well as external memory. The fact that neighbors access the same read-only data can be exploited - all local memories can be seen as a large distributed cache. In order to have a consistency mechanism that is easier to analyze, we can consider a weaker memory model: the parent tile aggregates data reported by children and all children tiles have read-only access to it.
Chapter 4

The Spawn System

A computation that consists of traversing or searching a tree can be executed in parallel by allowing processors to delegate subtree computation to other processors. The Spawn system follows this framework while trying to optimize communication for a 2D array of processors: work delegation is allowed only between adjacent processors, the tree being dynamically embedded into the 2D array.

The Spawn system implements the delegation mechanism by allowing function calls to be postponed and executed by another processor. Each processor generates a stack with jobs that can be reassigned: adjacent processors can be assigned jobs at the top of the stack; the local processor executes the jobs at the bottom of the stack (see Fig. 4-3). This policy ensures reassigned jobs are large - the first jobs spawned correspond to the top levels of the tree.

New jobs are created by spawning functions - calling instrumented versions of a function that essentially create a job on the spawn stack. Synchronization points can be used to wait for all functions spawned since the last sync point. A more detailed description of the spawn mechanism is presented in Section 4.3, and Section 4.4.

I will first present a simple Fibonacci example for Spawn; then I will illustrate how a binary heap traversal is embedded into the 2D mesh. Finally I will describe the C interface, and the inner mechanisms of the Spawn system.
/* Define an instrumented version of fib - spawn_fib */
spawndefine (fib, int, int, int);

/* Compute Fibonacci recursively */
void fib (const spawn_void_t* dummy, int* x, int* result)
{
    int xnew, res[2];
    if (*x <= 1)
    {
        *result = 1;
    }
    else
    {
        /* Recursion: 2 children are spawned */
        xnew = *x-1;
        spawn_fib (0, &xnew, &res[0]);
        xnew = *x-2;
        spawn_fib (0, &xnew, &res[1]);

        /* Synchronization: wait for both children to return */
        spawn_sync ();

        /* Use children’s results only after sync */
        *result = res[0] + res[1];
    }
}

Figure 4-1: Definition of a “spawnable” function: fib

4.1 Fibonacci Example

Figure 4-1 shows a recursive Fibonacci function that spawns its recursive calls:

1. an instrumented function wrapper is defined: spawn_fib (line 1)

2. 1st argument is ignored (line 5), since fib doesn’t require state data.

3. 2nd argument x holds the argument to the Fibonacci function (lines 5, 9, 16)

4. 3rd argument result will hold the result of the computation (lines 11, 2)
5. if computation is recursive, two children are spawned (lines 17, 19). Each child writes its result to a different temporary location, \( \text{res}[0,1] \).

6. a sync point waits for spawned children to finish (line 22)

7. only after the sync point the partial results can be used to compute the final result (line 25).

Note that it is not necessary to have all spawn calls and sync points in the same function body; these are C functions calls which are independent of context.

### 4.2 Heap Traversal Embedding

Figure 4-2: Processor state history: traversing a 1023 size heap on \( 4 \times 4 \)

A computation tree is embedded into the mesh by starting with the middle tile as the root of the tree. All edges of the tree correspond to links to adjacent tiles, or to the same tile. Figure 4-2 shows the embedding of a binary heap traversal into a \( 4 \times 4 \) tile array\(^1\). We notice how the children of a node are expanded to the neighbor tiles, or to the same tile. Symbol description:

* marks new expansions

\(^{1}\)the output is produced by running a sample program using the pthreads Spawn system; see Appendix F
marks nodes waiting for remote children to finish
marks idle tiles

num marks a tile working on heap node num. As for any binary heap, the children of x are 2x and 2x + 1

4.3 C Interface

Implementing the Spawn system can be done by writing a compiler, or by adding an extension to an existing language. My approach was to write a C library interface that defines the main operations required by the system. An implementation of the interface needs to define the following core primitives:

spawn_define (fname, ct_in_t, in_t, out_t) is a macro defining a spawnable version of the function given as first argument. An instrumented version of the original function fname is defined as a new function symbol spawn_fname. The new function has the same signature as the original function; at the same time fname is required to be defined with the following signature:

void fname () (const ct_in_t*, in_t*, out_t*)

spawn_define_callback (fname, ct_inAt, in_t, out_t, agg_t) is similar to spawn_define but all resulting values are aggregated to a single value by a specialized function. These updates are atomic with respect to function returns. The same result is achieved if all spawned children write to a vector of return values which is aggregated after all values are reported.

spawn_sync () adds a synchronization point, waiting for all spawns from the current sync frame to finish. A sync frame consists of all code executed between two consecutive sync points; entering a spawned function is considered a sync point.

We call a spawnable function a function that can be given as the first argument to spawn_define. The meaning of its arguments are:

\[2\text{additional primitives are specific to each implementation / system, and will be discussed in the corresponding section}\]
1. The first argument of type `ct.in.t*` is a pointer to a data structure that is not changed throughout the current sync frame. The system assumes `sync()` is called before this data is changed. As a result, no copies of this structure are made when the function is spawned.

2. The second argument of type `int*` is pointer to a data structure that can be changed by the children as well as by the parent. Each time a function is spawned, a copy of this structure is made; if the function is executed immediately, the copy should be made on the stack.

3. The third argument of type `out.t*` is a pointer to a data structure where the result is written when the function call returns. The parent can access this structure only after the first sync point after the spawn function call.

Two implementations for the Spawn interface were developed as part of the thesis project:

**Pthreads Spawn** creates $n \times n$ threads that communicate to each other as if they were a 2D mesh. They share the same memory space, and can use memory locks. An operating system with multi-processor support can schedule the threads to run in parallel on different processors. A simulation of any $n \times n$ configuration can be performed if sleep cycles are added as leaf and node computation.

**RAW Spawn** takes advantage of the $4 \times 4$ MIPS processors to schedule work in parallel on RAW. The system doesn’t implement locks, or shared memory and cache coherency protocols; these protocols can be enforced at user level. Real-time simulations of performance are available using the RAW simulation tools.

Semantically the Spawn framework is similar to a subset of the Cilk language developed at LCS. Its implementation has some draw-backs - efficiency, memory footprint, and some advantages - portability, simplicity. Being a C library, the pthreads implementation can be recompiled on any POSIX system. The RAW library works close to the hardware, implementing fast messaging and using limited signaling and no locking.
4.4 The Spawn Stacks

Each tile maintains a local stack with jobs that can be scheduled to run on adjacent tiles, or on the tile itself; these are called spawn stacks (Fig. 4-3). Most of the additional work done by spawn function calls consists of accessing data on the local spawn stack, or executing functions on the fly. On the other hand, idle tiles communicate with neighbor tiles trying to steal jobs from their spawn stacks. The following actions update the spawn stack:

**local execution** This is the most frequent case of spawn function execution. After the spawn stack size limit is reached, function calls are executed on the fly. Such executions are the most efficient, using the C local stack to save data. This action will suspend the current sync frame, and will push a new sync frame on the local stack.

**saved spawn** When a spawn function is executed and the spawn stack has empty slots, a spawn function call will save its state on the stack and return. The saved state provides data necessary for a later local or remote execution. The size of this state is not much larger than the size of the int function argument.

**local steal** When the local processor reaches a sync point, it will wait for the remote executions to finish, if any. However, if there are available jobs on the stack these will be stolen and executed locally; such executions are similar to local

![Figure 4-3: Heap traversal: work scheduling on 2 x 2 tiles](image)

36
executions.

**remote steal** If an idle tile succeeds in stealing a job from another tile’s spawn stack, the saved spawn state is transferred and the idle tile starts execution. As expected, the original owner of the job will not execute it; instead, sync points might wait for the stolen job to finish.

Taking Fig. 4-3 as an example, Processor A spawns 2, 3; it locally steals 2 and spawns 4, 5; it locally steals 4 and returns; at node 2 it waits for 5 to finish; at node 1 it waits for 3 to finish. Processor B steals remotely 3 from A and spawns 6, 7; and so on until at node 3 it reports results to A and becomes idle.

From the programmer’s perspective a spawned function call behaves like a regular function call which is executed concurrently with the current sync frame. The spawn function call is guaranteed to return and write its results by the time the control passes the first sync point that follows the spawn. Dealing with inter-processor communication, spawn stack management, and work scheduling are hidden in the spawn layer. On the other hand, all reads and writes done by spawned functions can happen concurrently. Unless locks are used for protection, exclusion policies have to be enforced at user level.

### 4.5 Shared Memory

One notices that the Spawn interface doesn’t address shared memory support. Different machines implement various memory models; it is extremely difficult, if not impossible to define a C interface that guarantees the same properties on any machine. However, if the user has access to a shared memory space, he or she can follow the memory consistency rules for a DAG of computation as described in [5]. Briefly, this policy requires that two threads that might be executed concurrently don’t read and write, or write and write at the same memory location; sync points should be used to mutually exclude these operations. In particular, such two threads are two children spawned in the same sync frame, or the parent and an active spawned child.
The above exclusion policy can be avoided by using standard locking mechanisms. However, deadlocks might occur in such conditions. A good way of avoiding deadlocks is holding no more than one lock at the same time by each thread.

If for some SMPs shared memory is fully implemented in hardware, other machines need additional help from the Spawn system. For instance, RAW Spawn flushes a child’s cache lines before returning, and it invalidates the parent’s cache lines after a return. It is the responsibility of the user to understand each system’s particularities, and design proper shared memory policies.
Chapter 5

RAW Spawn

The RAW implementation of Spawn is designed to run on the RAW chip and requires no underlying software layer. After the boot sequence, the Spawn system starts running. The user application has to define an entry point function similar to main() that starts executing on the root tile; calling spawn functions will ensure that work is assigned to the other RAW tiles. Most RAW libraries can still be used, unless they interfere with interrupts used by Spawn.

5.1 Interface Specification

spawn_define(fidx, fname, ct_in_t, in_t, out_t) is a macro defining a spawnable version of fname. See Section 4.3 for details. The only difference is the addition of the first argument; fidx an integer representing the position of fname in the spawn_register_fn declaration.

spawn_register_fn (fname1, ...) registers all the spawn functions which are defined using spawn_define. See Appendix B for examples using these macros.

spawn_sync() adds a synchronization point, waiting for all spawns from the current sync frame to finish. See Section 4.3 for details.

spawn_inton(), spawn_off() are primarily used by the Spawn system to commit atomic changes to the spawn stack - both interrupts and the main execution
thread access it. A critical code sequence can be inserted between intoff -inton to disable stealing/reporting, acting as a system-wide lock. This way the main thread can access data written by spawned functions before the sync point.

spawn_main() is a function that has to be defined by the application. After the spawn system boots up, this function is called on the root tile starting the user program.

raw.h defines miscellaneous functions specific to the raw system: raw_test_fail/pass/done, print_int/string, raw_get-cycle, etc. These functions can be used by applications as long as they don't interfere with interrupt handlers and messages used by the Spawn system.

5.2 Design Description

5.2.1 Messages

Stealing and reporting between tiles is implemented using the gdn dynamic network, and interrupting messages. All received messages trigger an interrupt handler that takes a certain action depending on the type of message, and the phase of a tile's life cycle (Table 5.1).

Each tile has a main thread of computation that follows the three phases described in Section 5.2.2. Interrupts are enabled only at certain times - a message might have to wait before being serviced. Section 5.2.3 shows why all messages are eventually read, and no deadlock can occur.

<table>
<thead>
<tr>
<th>type</th>
<th>description</th>
<th>content</th>
<th>phase I action</th>
<th>phase II action</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>try to steal</td>
<td>none</td>
<td>loop, send M2</td>
<td>interrupt-steal, send M2 or M3</td>
</tr>
<tr>
<td>M2</td>
<td>reject steal</td>
<td>none</td>
<td>wait, send M1</td>
<td>FAIL</td>
</tr>
<tr>
<td>M3</td>
<td>accept steal</td>
<td>fn arguments</td>
<td>loop, phase II</td>
<td>FAIL</td>
</tr>
<tr>
<td>M4</td>
<td>report results</td>
<td>fn results</td>
<td>FAIL</td>
<td>interrupt-report</td>
</tr>
</tbody>
</table>

Table 5.1: RAW spawn message types
5.2.2 Tile Life Cycle

Each tile’s life cycle consists of three phases. After boot, all but the root tile start in phase I. The latter starts in phase II; when it exists the computation has finished or an error has occurred.

During each phase the gdn interrupt handler routes incoming messages differently; transitions between phases require that interrupts are disabled. Fig.5-1 presents what operations happen at each phase, and how transitions occur.

![Diagram of Tile Life Cycle]

Figure 5-1: The 3 phases of a RAW tile: stealing, working, and reporting

I: Try Steal A tile is idle and it tries to steal work. All messages are received by the interrupt handler and dealt with immediately: M1 messages set a flag that asks the main thread to respond to the sender tile; M2 and M3 messages are recorded for later use; M4 messages generate an error.

The main thread loops, checking flags and sending messages: it responds to all M1 messages received with M2; it sends M1 messages, then it waits until M2 or M3 answers are received. If M2 is received, M1 is resent after a customizable number of loop cycles. If M3 is received, it jumps to phase II after all M2 replies have been sent. Note all M1 messages have been replied to by the time the transition happens. (see pseudo-code in Appendix C)
II: Run Spawn A tile works on a job and it allows other tiles to steal jobs. The main thread runs the computation, disabling interrupts when accessing the spawn stack. When the job ends it jumps to phase III. A job should not end unless all stolen jobs have reported back.

All messages are received and sent by the interrupt handler. If M1 is received, interrupt_steal() is called: M2 is sent if there are no available jobs on the spawn stack; otherwise M3 is sent along with the spawn function arguments. If M4 is received, interrupt_report() is called: function results are written to local memory. Receiving M2 or M3 generates an error, since no M1 message was sent.

III: Report Results A tile has finished a job and it reports the results. All interrupts are disabled; the incoming messages will be dealt with by the next phase I.

The main thread sends a M4 message to the parent, then it jumps to phase I.

5.2.3 Deadlock Avoidance

We will show that messages are sinkable throughout all 3 phases. This is sufficient to show deadlocks cannot occur (see Section 2.1.2).

In phase I the interrupt handler either sets a flag or it records the message, therefore messages are sinkable.

In phase II, M4 messages are sunk; M1 messages call interrupt_steal(). This handler returns M2 or M3 to the requesting tile, then it returns. The requesting tile is in phase I, and it will sink this message. No network congestion can be caused by a 3rd party, since only neighbor tiles talk to each other. Therefore interrupt_steal always returns, sinking the message.

In phase III, a M4 message is sent to the parent tile. Parent is in phase II, and will sink the message. All messages received during this period are postponed until next phase I sinks them.
5.2.4 RAW-Spawn Interface

The messaging protocol described so far is generic: a new system can be built if each tile defines means of executing and generating jobs. This allows a very efficient implementation for particular problems. All functions marked with † need to be implemented by the above software layer.

This interface uncouples the mechanism of messaging and stealing for 2D RAW meshes from the software layer implementing spawn stacks, sync points, and spawn function wrappers.

void spawn_run (void* msg, int* msg_len)† runs a stolen job taken from its argument. The result should be written to the same location before exiting.

void spawn_main ()† entry point of program

int spawn_interrupt_steal (void* msg, int* msg_len)† writes a new job to be sent to another tile; use NULL if not available.

void spawn_interrupt_report (void* msg, int* msg_len)† reads the result of a stolen job

void spawn_inton () turns interrupts on

void spawn_intoff () turns interrupts off

void spawn_sync_wait () waits until spawn_sync_signal is called. Interrupts are disabled before calling the function and upon exit.

void spawn_sync_signal () causes spawn_sync_wait to exit

5.2.5 Spawn Layer

This layer is written in C, allowing portability. The most important component is the spawn stack: sync frames and spawn calls are pushed or popped from the stacks. Each spawn call corresponds to a sync frame; the stack keeps track of how many spawn calls are executed locally or remotely for each sync frame. The spawn stacks are used
by spawn_fn to add jobs; by spawn_sync to execute jobs and query sync frame status; by spawn_interrupt_steer to remove jobs; and by spawn_interrupt_report to update sync frame status.

All accesses to data structures shared by the main thread and interrupt handlers are controlled by disabling and enabling interrupts; this protocol is similar to acquiring and releasing a system wide lock: intoff acquires the lock, inton releases it. The wait - signal mechanism is used when a sync point waits for remote tiles to report data.

### 5.2.6 Efficiency

In terms of memory usage, the spawn stack uses 12 additional bytes for each saved function call. Unfortunately, the CPU takes significantly more cycles due to aligning data, following pointers, updating sync frame status, etc. Consequently, spawning functions is relatively expensive compared to calling native C functions (Section 5.3 estimates a slowdown factor of 2-3).

CPU performance could be improved if function wrappers are written directly in assembly. The best performance could be achieved if code is automatically generated by an auxiliary program. Moreover, the code could be suited to each function’s arguments, no longer requiring a fixed signature format.

There are two optimizations that speedup spawn calls: i) spawn calls that exceed the stack limit are executed directly on the C stack; ii) after a certain depth level all spawn and sync statements are replaced with their C equivalent - in particular, there is no spawn spawn stack activity, and no possibility to steal work beyond this level.

Repeatedly trying to steal is another reason for overhead. Each time the handler is called a constant cost is paid - saving and restoring registers, querying the spawn stack, etc. Customizing the frequency of steal requests affects both raw performance and load balancing.

Finally, stealing and reporting adds a constant overhead. If the granularity of the stolen job is not sufficiently large, the steal-report overhead can become significant.
5.3 Performance Analysis

The overheads of the RAW Spawn system are first analyzed in this section. A speedup benchmark is later performed, showing how problem size and work granularity influence speedup for a given mesh size.

5.3.1 Tests

Traversals of two tree configurations are used as benchmarks: i) bin - a perfect binary tree having all leafs at the same level; ii) fib - the Fibonacci tree. These tests recursively count the number of leafs, and compare it with a precomputed number; if the numbers don’t match, the test fails. Each test case takes the following parameters:

1. name - fib, bin
2. n - depth of binary tree, or fib argument
3. stealwait - number of loops waited before resending M1 (approx. 40 cycles/loop)
4. Clevels - the depth of leaf subtrees computed in C
5. spawnlevels - the maximum depth of the spawn stack
6. mesh size

5.3.2 Terminology

Work granularity is defined as the smallest job that can run on a single tile. Looking at Appendix B we notice that the spawn function calls a native C function when the argument is less than Clevels. In other words, only n – Clevels levels are available to Spawn, and the leaf computation is a subtree of size Clevels.

Parallelism is defined as \( P = T_i / T_\infty \), where \( T_i \) represents ideal runtime on \( i \) processors. Considering how Clevels affects work granularity, we have \( P_{\text{bin}} = 2^{n-C\text{levels}} \) and \( P_{\text{fib}} < fib(n - C\text{levels} + 1) \). If spawn levels < n – Clevels parallelism is further decreased.

Efficiency is defined as \( E = T_i / PT_P \), where \( T_i \) is running time on \( i \) processors, and \( P \) is the number of processors.
### Table 5.2: Average leaf work - Bin

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C cycles</th>
<th>cycles 1 leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>10</td>
<td>39,256</td>
<td>38.33</td>
</tr>
<tr>
<td>A2</td>
<td>12</td>
<td>156,056</td>
<td>38.09</td>
</tr>
<tr>
<td>A3</td>
<td>13</td>
<td>311,736</td>
<td>38.05</td>
</tr>
<tr>
<td>A4</td>
<td>14</td>
<td>623,064</td>
<td>38.02</td>
</tr>
<tr>
<td>A5</td>
<td>15</td>
<td>1,245,688</td>
<td>38.015</td>
</tr>
</tbody>
</table>

### Table 5.3: Average leaf work - Fib

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C cycles</th>
<th>cycles 1 leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>10</td>
<td>3,960</td>
<td>44.49</td>
</tr>
<tr>
<td>G2</td>
<td>12</td>
<td>9,928</td>
<td>42.60</td>
</tr>
<tr>
<td>G3</td>
<td>15</td>
<td>40,938</td>
<td>41.47</td>
</tr>
<tr>
<td>G4</td>
<td>20</td>
<td>449,417</td>
<td>41.05</td>
</tr>
<tr>
<td>G5</td>
<td>26</td>
<td>8,053,961</td>
<td>41.00</td>
</tr>
</tbody>
</table>

### 5.3.3 Base Case

By using the C only function on $1 \times 1$, the $Cycles$ base case is computed for several mesh sizes. Since the RAW simulator has a constant overhead for any mesh size, it is hard to compute base cases for very large $n$. Table 5.2 and Table 5.3 show the number of $C$ cycles per leaf node. We notice that this number converges as problem size increases. It is safe to assume for large $n$ we can compute $Cycles_{bin}(n) = 38.015 \times 2^n$, and $Cycles_{fib}(n) = 41.0 \times fib(n)$. All values computed this way are marked with $\dagger$.

### 5.3.4 Spawn Overheads

Overall speedup is influenced by the following factors:

1. constant cost overheads: spawn function calls, interrupting try-steal messages (M1), accept-steal and report-results messages (M3, M4)

2. improper load balancing: bad tree embedding, lack of parallelism i.e. $n - Clevel$ is too small

---

$^1P = 2^{2n}$ is typically needed for a $n \times n$ mesh. See Section 6.3
5.4: Overhead of spawn function calls

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C</th>
<th>spawn speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>levels</td>
<td>cycles</td>
</tr>
<tr>
<td>B1</td>
<td>14</td>
<td>1</td>
<td>1,262,359</td>
</tr>
<tr>
<td>B2</td>
<td>14</td>
<td>4</td>
<td>703,159</td>
</tr>
<tr>
<td>B3</td>
<td>14</td>
<td>7</td>
<td>633,207</td>
</tr>
<tr>
<td>B4</td>
<td>14</td>
<td>10</td>
<td>624,439</td>
</tr>
</tbody>
</table>

Table 5.4: Overhead of spawn function calls - Bin

5.5: Overhead due to steal requests

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C</th>
<th>spawn speedup</th>
<th>steal every</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cycles</td>
<td>1 × 1</td>
<td># cycles</td>
</tr>
<tr>
<td>C1</td>
<td>15</td>
<td>10</td>
<td>1,616,001</td>
<td>0.77</td>
</tr>
<tr>
<td>C2</td>
<td>15</td>
<td>25</td>
<td>1,425,025</td>
<td>0.87</td>
</tr>
<tr>
<td>C3</td>
<td>15</td>
<td>50</td>
<td>1,342,254</td>
<td>0.93</td>
</tr>
<tr>
<td>C4</td>
<td>15</td>
<td>100</td>
<td>1,296,142</td>
<td>0.96</td>
</tr>
<tr>
<td>C5</td>
<td>15</td>
<td>200</td>
<td>1,271,435</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Table 5.5: Overhead due to steal requests - Bin

5.6: Constant cost overheads for $\tilde{P} = 2^3$ - Bin

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C</th>
<th>steal</th>
<th>spawn speedup</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cycles</td>
<td>wait</td>
<td>1 × 2</td>
<td>3 × 3</td>
</tr>
<tr>
<td>D1</td>
<td>13</td>
<td>10</td>
<td>200</td>
<td>311,736</td>
<td>1.87</td>
</tr>
<tr>
<td>D2</td>
<td>14</td>
<td>11</td>
<td>200</td>
<td>623,064</td>
<td>1.93</td>
</tr>
<tr>
<td>D31</td>
<td>15</td>
<td>12</td>
<td>200</td>
<td>1,245,688</td>
<td>1.96</td>
</tr>
<tr>
<td>D32</td>
<td>15</td>
<td>12</td>
<td>150</td>
<td>1,245,688</td>
<td>6.21</td>
</tr>
<tr>
<td>D33</td>
<td>15</td>
<td>12</td>
<td>100</td>
<td>1,245,688</td>
<td>6.64</td>
</tr>
<tr>
<td>D34</td>
<td>15</td>
<td>12</td>
<td>75</td>
<td>1,245,688</td>
<td>3.74</td>
</tr>
<tr>
<td>D35</td>
<td>15</td>
<td>12</td>
<td>50</td>
<td>1,245,688</td>
<td>7.15</td>
</tr>
<tr>
<td>D36</td>
<td>15</td>
<td>12</td>
<td>25</td>
<td>1,245,688</td>
<td>7.08</td>
</tr>
<tr>
<td>D41</td>
<td>17</td>
<td>14</td>
<td>200</td>
<td>4,982,752$^\dagger$</td>
<td>1.99</td>
</tr>
<tr>
<td>D42</td>
<td>17</td>
<td>14</td>
<td>100</td>
<td>4,982,752$^\dagger$</td>
<td>7.61</td>
</tr>
<tr>
<td>D43</td>
<td>17</td>
<td>14</td>
<td>75</td>
<td>4,982,752$^\dagger$</td>
<td>3.92</td>
</tr>
<tr>
<td>D44</td>
<td>17</td>
<td>14</td>
<td>50</td>
<td>4,982,752$^\dagger$</td>
<td>7.56</td>
</tr>
<tr>
<td>D45</td>
<td>17</td>
<td>14</td>
<td>25</td>
<td>4,982,752$^\dagger$</td>
<td>7.30</td>
</tr>
<tr>
<td>D5</td>
<td>20</td>
<td>17</td>
<td>200</td>
<td>39,862,016$^\dagger$</td>
<td>3.97</td>
</tr>
</tbody>
</table>

Table 5.6: Constant cost overheads for $\tilde{P} = 2^3$ - Bin
Table 5.4 shows how the overhead of spawn function calls decreases as granularity increases. The base case is compared to Spawn running on $1 \times 1$; since only the root tile runs, there are no interrupting messages that come with their own overhead. The maximum slowdown factor is 2, but for $C_{levels} = 10$ there is virtually no overhead. The corresponding granularity is 40K cycles (see Table 5.2).

The overhead generated by steal interrupts is tested in Table 5.5. The root tile runs the C version in a $3 \times 3$ configuration. As a result all neighbor tiles unsuccessfully send try-steal messages. As expected, we notice that overhead decreases as steal wait increases. For values 50 - 200 the overhead is small. Moreover, in practice not all neighbor tiles try stealing and fail, and keep trying again.

The experiment from Table 5.6 keeps $P = 8$ for all tests. Therefore, all factors that cause bad performance are analyzed, except parallelism. The $2^3$ tree can easily be embedded into the $3 \times 3$ and $4 \times 4$ meshes assigning one leaf to a distinct tile. Subsequently, we are expecting a speedup of 8, unless we have bad tree embeddings.

For $4 \times 4$ we notice that increasing granularity up to 14 helps. At the same time, best performance is achieved for $stealwait = 50 - 100$. This tendency becomes more obvious for $3 \times 3$: $stealwait = 100 - 200$ drastically affects load balancing, probably due to bad embeddings. We conclude $stealwait = 50$ is most suitable, and $C_{level} = 14$ virtually removes constant cost overheads - 7.58 (D44). $C_{level} = 10$ is doing almost as well - 7.04 (D35).

### 5.3.5 Parallelism

We have established that sufficiently large granularity diminishes constant system overheads. We will study how parallelism influences speedup for various mesh sizes. In most of the following experiments $stealwait = 50$ and $spawnlevels = 10$.

Table 5.7 illustrates how increasing parallelism for constant granularity improves speedup. $C_{levels} = 14$ was chosen to ensure that constant overheads don't play a big role. For $4 \times 4$ speedup is not much improved after $n > 24$, $\tilde{P} = 2^{10}$. (E34,

---

2 ideally a tree is randomly embedded like a water drop that "spills" on a surface. Large steal wait periods might generate artifacts on small meshes
### Table 5.7: Increasing parallelism for constant granularity - Bin

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C levels</th>
<th>C wait</th>
<th>C cycles</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>E11</td>
<td>17</td>
<td>14</td>
<td>200</td>
<td>4,982,752</td>
<td>3.87</td>
</tr>
<tr>
<td>E12</td>
<td>19</td>
<td>14</td>
<td>200</td>
<td>19,931,008</td>
<td>5.27</td>
</tr>
<tr>
<td>E21</td>
<td>17</td>
<td>14</td>
<td>100</td>
<td>4,982,752</td>
<td>7.61</td>
</tr>
<tr>
<td>E22</td>
<td>19</td>
<td>14</td>
<td>100</td>
<td>19,931,008</td>
<td>6.29</td>
</tr>
<tr>
<td>E23</td>
<td>21</td>
<td>14</td>
<td>100</td>
<td>79,724,032</td>
<td>7.45</td>
</tr>
</tbody>
</table>

### Table 5.8: Tradeoff between granularity and parallelism - Bin

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C levels</th>
<th>C wait</th>
<th>C cycles</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>20</td>
<td>8</td>
<td>50</td>
<td>39,862,016</td>
<td>3.93</td>
</tr>
<tr>
<td>F2</td>
<td>20</td>
<td>9</td>
<td>50</td>
<td>39,862,016</td>
<td>3.94</td>
</tr>
<tr>
<td>F3</td>
<td>20</td>
<td>10</td>
<td>50</td>
<td>39,862,016</td>
<td>3.95</td>
</tr>
<tr>
<td>F4</td>
<td>20</td>
<td>11</td>
<td>50</td>
<td>39,862,016</td>
<td>3.97</td>
</tr>
<tr>
<td>F5</td>
<td>20</td>
<td>12</td>
<td>50</td>
<td>39,862,016</td>
<td>3.99</td>
</tr>
</tbody>
</table>

### Table 5.9: Tradeoff between granularity and parallelism - Fib

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C levels</th>
<th>C wait</th>
<th>C cycles</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>26</td>
<td>9</td>
<td>50</td>
<td>8,053,961</td>
<td>1.97</td>
</tr>
<tr>
<td>H1</td>
<td>26</td>
<td>12</td>
<td>50</td>
<td>8,053,961</td>
<td>7.84</td>
</tr>
<tr>
<td>H1</td>
<td>26</td>
<td>15</td>
<td>50</td>
<td>8,053,961</td>
<td>1.97</td>
</tr>
<tr>
<td>H1</td>
<td>26</td>
<td>18</td>
<td>50</td>
<td>8,053,961</td>
<td>7.86</td>
</tr>
<tr>
<td>test no.</td>
<td>n</td>
<td>C levels</td>
<td>steal wait levels</td>
<td>spawn levels</td>
<td>C cycles</td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>----------</td>
<td>-------------------</td>
<td>--------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>26</td>
<td>15</td>
<td>50</td>
<td>8</td>
<td>1,245,688</td>
</tr>
<tr>
<td>I2</td>
<td>26</td>
<td>15</td>
<td>50</td>
<td>10</td>
<td>1,245,688</td>
</tr>
<tr>
<td>I3</td>
<td>26</td>
<td>15</td>
<td>50</td>
<td>12</td>
<td>1,245,688</td>
</tr>
</tbody>
</table>

Table 5.10: Increasing spawn stack size - Fib

<table>
<thead>
<tr>
<th>test no.</th>
<th>n</th>
<th>C levels</th>
<th>steal wait levels</th>
<th>spawn levels</th>
<th>C cycles</th>
<th>speedup</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 x 3</td>
<td>4 x 4</td>
</tr>
<tr>
<td>J1</td>
<td>26</td>
<td>15</td>
<td>50</td>
<td>12</td>
<td>1,245,688</td>
<td>8.03</td>
<td>11.35</td>
</tr>
<tr>
<td>J2</td>
<td>28</td>
<td>15</td>
<td>50</td>
<td>13</td>
<td>21,085,543</td>
<td>8.46</td>
<td>12.04</td>
</tr>
<tr>
<td>J3</td>
<td>29</td>
<td>15</td>
<td>50</td>
<td>14</td>
<td>34,117,126</td>
<td>8.56</td>
<td>12.73</td>
</tr>
<tr>
<td>J4</td>
<td>30</td>
<td>15</td>
<td>50</td>
<td>15</td>
<td>55,202,669</td>
<td>8.46</td>
<td>12.76</td>
</tr>
</tbody>
</table>

Table 5.11: Increasing parallelism and stack size for constant granularity - Fib

35); for $3 \times 3$ $n = 21..24$, $\bar{P} = 2^7 - 2^9$ approaches the upper bound (E33, E34); for $2 \times 2$ speedup is perfect for $n = 17$, $\bar{P} = 2^3$. We notice again how large stealwait negatively influence speedup on $3 \times 3$ for small $P$ (E12, E22). On the other hand, comparing E4 to E21 we conclude that parallelism takes precedence over granularity.

Experiments described in Table 5.8 and Table 5.9 try to optimize speedup for a certain problem size $n$. We already know larger granularity and parallelism improve speedup, and $Clevel = 14$, $\bar{P} = 2^{10}$ give good results on $4 \times 4$. However, the user has to make a tradeoff if $n < 24$. For $n = 20$ F3, F4 show $\bar{P} = 2^9 - 2^{10}$ is the most important factor, not larger granularity. For fib the problem size is even smaller: $fib(26) = 2^{17} - 2^{18}$. Best performance is achieved for $Clevel = 12 - 15$ (10-40K cycles), and $\bar{P} = 200 - 700$ (H1).

The last experiments increase parallelism for the fib computation, keeping constant granularity. In Table 5.10 we study how spawnlevels affect performance; indeed increasing spawnlevels improves performance for constant problem size (I3). In Table 5.11 we see how increasing $n = 26..30$ improves speedup, which reaches its upper bound for $n = 29$, $\bar{P} = fib(14) - fib(15) = 2^9 - 2^{10}$

To summarize the results, we have concluded that i) constant system overhead can be avoided for granularity greater than 40K cycles ii) stealwait = 50 is the best setting for small and large problems; iii) for $4 \times 4$, $\bar{P} = 2^9 - 2^{10}$ gives a remarkable
speedup of $12 - 13$ for both tree traversals - bin(20) $C_{level} = 10$, fib(29) $C_{level} = 15$;

iv) if problem size is small, it is better to increase $\bar{P}$ as long as granularity is not smaller than $2^{10}$. 
Chapter 6

Pthreads Spawn

The pthread implementation of the Spawn system works on any POSIX system. The library creates pthreads corresponding to each virtual processor; on a real multiprocessor system these threads are typically mapped to individual processors, therefore utilizing the full capacity of the hardware.

6.1 Interface Specification

spawn_define(fname, ct_in_t, in_t, out_t) is a macro defining a spawnable version of fname. See Section 4.3 for details.

spawn_sync() adds a synchronization point, waiting for all spawns from the current sync frame to finish. See Section 4.3 for details.

spawn_init(int xdim, int ydim, int xinit, int yinit, int size, int depth, int len) initializes the Spawn system, setting the size of the 2D array, the coordinates of the starting tile, the maximum spawn stack size, the maximum stack depth, and the size of the labels. No spawn functions can be used before calling spawn_init.

spawn_cookie() returns a pointer to the string associated with the current tile.

spawn_cookie_len() returns the size of the cookie string; this size is specified when calling spawn_init.
**spawn_print_cookies()** prints a graphical representation of all tiles and their labels; this function can be used to illustrate what each processor is working on (Appendix F).

**spawn.done()** frees all resources used by the Spawn system; no spawn functions can be used after this point.

### 6.2 Design Description

#### 6.2.1 Memory and Synchronization

All data structures are shared by all threads but only adjacent threads interact with each other. Each thread uses a spawn stack and two locks with condition variables.

The ops.lock is used to control access to the spawn stack; the owner thread adds and removes spawned functions and sync frames. The only exception is when another thread does a remote steal.

The sig.lock is used to control signaling. When a thread tries to remote steal it locks its own signal lock, then it checks the surrounding ops.lock. If it steals a job it releases sig.lock, otherwise it waits for it. On the other hand, when a thread spawns a job it tries to signal this event to all neighbors: it grabs their sig.lock, then it signals them. This lock is also used at synchronization points where spawn.sync waits for remote steals to finish.

This implementation heavily relies on using mutexes for every operation. The most demanding task is signaling available spawns (8 mutex ops, 4 signals) and accessing local data structures (2 mutex ops). The advantage of this approach is that all threads are either working or waiting for work to become available. A pool based mechanism adds some lag, but it can be cheaper in terms of CPU for working tiles. Another design alternative is using interrupts rather then locks, similar to the RAW implementation.
6.2.2 Efficiency

The most common operations are spawning functions and synchronizing. The overhead of these functions is relatively high compared to the standard C function call: when spawning a copy of the 2nd argument is created, the spawn stack is locked and updated, etc. This impediment can be avoided if the spawn functions are used only at the higher levels of the tree, and regular C calls are used for the computation intensive leafs. The Spawn system limits the depth of the spawn stack, but the programmer needs to consider this issue as well.

A related issue is tailoring the stack size; a larger stack decreases chances that neighbor processors run out of work, but it increases memory footprint.

6.3 Performance Analysis

The pthread implementation of the Spawn system was tested by traversing two simple tree structures: i) a binary heap ii) a Fibonacci tree (Appendix F). The experiments were aimed at testing whether the system is running as planned, and evaluating the scalability of the Spawn framework on different processor configurations. Measuring the overhead generated by the pthreads system was not a priority.

All tests were performed on a single processor box running Linux. How is it possible to evaluate a multi-processor system on a single CPU machine? To solve this issue sleep cycles were added to the tree traversal: the same program runs slow motion, allowing the operating system to switch between threads. The sleep cycles are a few orders of magnitude larger than the real CPU usage. The shortcomings of this approach are not being able to
i) measure the overhead of the Spawn implementation;
ii) scale pthreads overhead with CPU number - only one CPU runs all threads;
iii) obtain a scheduling of threads similar to that of a multiprocessor machine.

Runtime speedup was computed by measuring the real runtime vs. the sum of all sleep cycles over the tree. Table 6.1 summarizes the speedups for traversing the binary heap using different leaf/node sleep cycles and mesh sizes.
Table 6.1: Speedup for binary heap traversal

Experiments A1-A4 aim at determining the magnitude of the sleep cycles. For \( leaf > 100\text{ms}\) the performance is not affected; we assume 100 is reasonably large.

Experiments B1-B4, C1-C4 add node weights studying how different leaf/node weight ratios affect the speedup. For small heaps, speedup is affected for any mesh size (predominantly for large \( n\)). For large heaps only large \( n\) are affected.

If \( U(n, heap - size, node - leaf - ratio) = \frac{speedup}{n^2} \) measures the CPU utilization, we can say \( \frac{\partial U}{\partial n} < 0, \frac{\partial U}{\partial heap - size} > 0, \) and \( \frac{\partial U}{\partial node - leaf - ratio} < 0. \)

Experiments D1-D12 measure how speedup is affected by problem size. I assumed the 1/10 leaf/node weight ratio is realistic\(^1\). We notice that speedup increases with

\(^1\)the program should be designed to minimize Spawn overhead; moreover, Spawn helps by limiting the depth of the spawn stack
heap size for any mesh size, reaching an upper bound at large heap sizes. For $4 \times 4$ the limit is $13$ at $2^{10}$, for $6 \times 6$ the limit is $26$ at $2^{12}$, for $8 \times 8$ the limit is $50$ at $2^{16}$. These numbers suggest a speedup of $0.75n^2$ is achieved for problem size $2^{2n}$. A $4^n$ tree has 4 children and depth $n$; the tile graph has maximum distance $n$ (middle to corner), while each tile has 4 neighbors. It is easy to prove this correlation for a 1D mesh when the root is one end (there is always one possible steal). Using statistical arguments the same could be proved for 2D.

Experiments E1-E5 from Table 6.2 show similar speedup limits for the Fibonacci tree traversal\(^2\). This gives us hope the above experiments are representative for a larger class of problems, and the Spawn framework could work successfully for most search problems.

Future testing needs to be done on a multi-processor machine. Such tests would show what low level optimizations are required to make the pthreads Spawn system effective in terms of real speedup.

\(^2\)100ms leaves, 10ms nodes
Chapter 7

Conclusion

This thesis has presented a system that exploits coarse grain parallelism on 2D processor meshes. Experimental results proved the system achieves good performance on the RAW chip. Moreover, a simulation of large processor meshes showed constant efficiency up to $12 \times 12$.

A C library has been developed, allowing applications to spawn function calls and wait at synchronization points. As opposed to parallel languages that use specialized compilers, the Spawn system uses the existing C infrastructure. This approach allows better portability, and simplicity of use. On the other hand, efficiency of spawn function calls becomes an issue. Experimental results have shown that work granularity of 40K cycles highly diminishes constant cost overheads.

The two benchmarks used for testing are traversals of perfect binary trees, respectively Fibonacci trees. These tests showed good speedups for the RAW chip, as well as for the pthreads simulation. On RAW, speedups of $12 - 13$ are achieved for $4 \times 4$ when parallelism exceeds $2^9 - 2^{10}$; for $3 \times 3$ speedups of $7 - 8$ are obtained for problems with parallelism $2^7 - 2^9$. Moreover, pthreads simulation results showed speedups of $35 - 50$ for $8 \times 8$, $95 - 110$ for $12 \times 12$, and $120 - 150$ for $16 \times 16$. Empirically, we can generalize that a speedup of $0.75n^2$ is achieved for $n \times n$ meshes when parallelism reaches $2^{2n}$. If this is correct, the Spawn computation model is efficient.

For large processor meshes point to point communication is the only alternative. Relying on main memory is no longer possible due to scalability issues. The natural
tendency is keeping data and computation local; as communication is point to point, work partitioning needs to be local in order to maintain locality of data access. Dynamically embedding computation trees into 2D processor meshes is an alternative that reaches this goal.

7.1 Future Work

The Spawn system for RAW has constant cost overheads incurred when spawning function calls, and synchronizing. Efficiency can be improved by automatically generating C code that is tailored to each spawn function signature. Considering the capabilities of modern compilers, saving data on the spawn stack should be no slower that pushing data on the C stack. Another optimization is using fast interrupt handlers when no jobs are available on the stack - there is no need to save registers, since the code path is simple for this case.

There are several possible extensions to the RAW Spawn library: i) an initialization function that sets up configuration at runtime - spawn stack size and depth, steal frequency, and C-level depth; ii) a define_callback macro that aggregates data when calls return, avoiding the hassle of keeping vectors with results; iii) support for locking and shared memory allocation, allowing the implementation of more complex algorithms such as scientific computing.

Quantifying system overheads and analyzing how they affect performance would help choosing optimal configuration parameters. A related research problem is proving a statistical lower bound on speedup given mesh size, tree structure, and problem size.
Appendix A

RAW Spawn Interface

```c
#ifndef SPAWN_H
#define SPAWN_H
#ifdef __cplusplus
extern "C" {
#endif
#include "spawn-c2asm.h"
#ifndef SPAWN_OFF
#include "spawn-stack.h"
#endif SPAWN_OFF
#include "spawn_stack.h"

/ *
 * INIT
 */
static inline void spawn_init ()
{
    pinit ();
    spawn_stack_init ();
    spawn_stack_push_sync ();
}

static inline void spawn_done ()
{
    spawn_stack_pop_sync ();
}

/ *
 * SYNC
 *
 * Adds a synchronization point, waiting for all spawns from the
 * lowest sync frame to finish. Possible actions:
 * - steal locally
 * - wait for all remote spawns to finish
 */
static inline void spawn_sync ()
{
    spawn_t *sp;
    if (spawn_stack_cexec) return;
    spawn_intoff ();
```
```c
#include DEB_SPAWN
pstr("syncB "); spawn_stack_print();
#endif

// loop & local steal
while ((sp = spawn_stack_local_steal_push_sync ()))
{
    spawn_inton ();

    if ( SPAWN_IN_NULL(sp->flags) )
        (*spawn_functions[sp->fn]) (sp->ct_, 0, sp->out_);
    else
        (*spawn_functions[sp->fn]) (sp->ct_, sp->data, sp->out_);

    spawn_intoff ();
    spawn_stack_local_steal_pop_sync ();
}

// wait for remote steals
while (!spawn_stack_sync_is_done ())
    spawn_sync_wait ();

if (spawn_stack_sync_is_remote ())
    spawn_invalidate_cache ();

#endif
spawn_inton ();
}

*/
* SPAWN
*
* spawn_define & spawn_callback_define are macros defining a
* spawnable version of the function given as argument. A new function
* symbol is defined - spawn_fn - where fn is the original name of the
* function.
*
* Forward declaration :
* void fn_name (const ct_t*, int_t*, out_t*);
* *
* spawn_fn: write result to local frame; no access allowed to out_t
* before sync().
* spawn_define (fn_name, ct_t, in_t, out_t)
* => spawn_fn_name (const ct_t*, in_t*, out_t*)
*
* Possible actions of spawn (used by spawn_fn, spawn_callback_fn):
* - C execute
* - spawn execute (if the policy preffers earlier spawns)
* - save spawn
*/
```
#define spawn_register_fn(fns...) spawnable_fn_t spawn_functions[] = {fns};

typedef struct {} spawn_void_t;

#define spawn_define(num, fnname, ct_t, in_t, out_t) \
  /* Forward declaration */ \
  void fn_name (const ct_t*, in_t *, out_t *); \
  /* Define custom function */ \
  inline void spawn_## fn_name (const ct_t *ct_a, in_t *in_a, out_t *out_a) \
  { \
    if (spawn_stack_cexec) \
      { /* fastest - C exec */ \
        if (sizeof(in_t) > 0 & & in_a) { \
          in_t saved_in = *in_a; \ 
          fn_name (ct_a, &saved_in, out_a); \
        } else \ 
          fn_name (ct_a, 0, out_a); \
      } \
    else if (spawn_stack_sexec) \
      { /* almost as fast - local exec - create sync on stack (no save) */ \
        spawn_intoff (); \ 
        spawn_stack_push_sync (); \ 
        spawn_inton (); \ 
        \
        if (sizeof(in_t) > 0 & & in_a) { \
          in_t saved_in = *in_a; \ 
          fn_name (ct_a, &saved_in, out_a); \
        } else \ 
          fn_name (ct_a, 0, out_a); \
        \
        spawn_intoff (); \ 
        spawn_stack_pop_sync (); \ 
        spawn_inton (); \ 
      } \
    else \
      { /* save spawn */ \
        spawn_intoff (); \ 
        SPAWN_STACK_SAVE_SPAWN (num, ct_t, ct_a, in_t, in_a, out_t, out_a); \ 
        \
        SPAWN_STACK_SAVE_SPAWN (num, ct_t, ct_a, in_t, in_a, out_t, out_a); \ 
        \
        spawn_intoff(); \ 
        /* pstr("spawn ")); spawn_stack_print(); */ \
        spawn_inton(); \ 
      }

#else // SPAWN-OFF, no stack

typedef struct {} spawn_void_t;

#define spawn_init() {}
#define spawn_done() {}
#define spawn_sync() {}
#define spawn_register_fn(fns...)

#define spawn_define(num, fn_name, ct_t, in_t, out_t)

/* Forward declaration */
void fn_name (const ct_t*, in_t*, out_t*);

/* Define custom function */
inline void spawn_##fn_name (const ct_t* ct_a, in_t* in_a, out_t* out_a) {
  if (sizeof(in_t) > 0 && in_a) {
    in_t saved_in = *in_a;
    fn_name (ct_a, &saved_in, out_a);
  } else {
    fn_name (ct_a, 0, out_a);
  }
}

#endif
#endif
#endif
// SPAWN_H
Appendix B

RAW Spawn Examples

Definition of Fibonacci function:

```c
#include "spawn.h"
#include "raw.h"

int fib_c(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib_c(n-1) + fib_c(n-2);
}
```

/* Compute Fibonacci recursively */
spawn_define(0, fib, spawn_void_t, int, int);
spawn_register_fn((spawnable_fn_t)&fib);

void fib (const spawn_void_t* dummy, int* x, int* result)
{
    int xnew, res[2];
    if (*x <= ARG_CLEVEL)
    {
        /* Use C function for last levels */
        *result = fib_c(*x);
    }
    else
    {
        /* Recursion: 2 children are spawned */
        xnew = *x-2;
        spawn_fib (0, &xnew, &res[1]);
        xnew = *x-1;
        spawn_fib (0, &xnew, &res[0]);

        /* Synchronization: wait for both children to return */
        spawn_sync ();

        /* Use children's results only after sync */
        *result = res[0] + res[1];
    }
}
int main()
{
    unsigned long long int t1, t2;
    int i, a = 1, b = 1;
    int n = ARG_N, res;

    /* Computing Fibonacci iteratively */
    for (i=0; i<n-1; i++) {
        b = a+b; a = b-a;
    }

    /* Computing Fibonacci recursively */
    t1 = raw_get_cycle();
    if (ARG_CLEVEL < 0) {
        /* C only */
        res = fib_c(n);
    } else {
        /* Spawn */
        spawn_fib (0, &n, &res);
        spawn_sync (;
    }
    t2 = raw_get_cycle();

    /* Check & report results */
    if (b != res) raw_test_fail (18999);
    pinit();
    pstr("RESULTS: fib("; pint(n); pstr("=") ; pint(res);
    pstr(" cycles="); pint((int)(t2-t1));
    pstats(); pstr("\n"); pflush();
    return 0;
}
Definition of binary tree traversal:

```c
#include "spawn.h"
#include "raw.h"

int bin_tree_c (int depth)
{
    if (depth <= 0)
        return 1;
    else
        return bin_tree_c (depth-1) + bin_tree_c (depth-1);
}

/* Count binary tree leafs recursively */
spawn_define(0, bin_tree, spawn_void_t, int, int);
spawn_register_fn((spawnablefn_t)&bin_tree);

void bin_tree (const spawn_void_t* dummy, int* depth, int* result)
{
    int res[2], ndepth;
    if (*depth <= ARG_CLEVEL)
        /* Use C function for last levels */
        *result = bin_tree_c(*depth);
    else
        /* Recursion: 2 children are spawned */
        ndepth = *depth - 1;
        spawn_bin_tree (0, &ndepth, &res[0]);
        spawn_bin_tree (0, &ndepth, &res[1]);

        /* Synchronization: wait for both children to return */
        spawn_sync (;

        /* Use children's results only after sync */
        *result = res[0] + res[1];
}

int main()
{
    unsigned long long int t1, t2;
    int n = ARG_N, res;

    /* Count the leafs of a binary tree recursively */
    t1 = raw_get_cycle();
    if (ARG_CLEVEL < 0) {
        /* C only */
        res = bin_tree_c (n);
    } else {
        /* Spawn */
        spawn_bin_tree (0, &n, &res);
        spawn_sync (;
```
}  
l2 = raw_get_cycle();

/* Check results, report settings */
if (res != (1<<n)) raw_test_fail (18999);
pinit();
pstr("RESULTS: bin_tree("); pint(n); pstr("="); pint(res);
pstr(" clevel="); pint(ARG_CLEVEL);
pstr(" cycles="); pint((int)(t2-t1));
pstats(); pstr("\n"); pflush();
return 0;
}
Appendix C

RAW Spawn Source Code Extracts

Spawn stack source code:

```c
#ifndef SPAWN_STACK_H
#define SPAWN_STACK_H
#include "spawn_c2asm.h"

#define SPAWN_MAX_SYNC_SPAWNS 4
#define SPAWN_MAX_SPAWNS SPAWN_MAX_SYNC_SPAWNS*SPAWN_MAX_SYNCS

#define SPAWN_MAX_FUNCTIONS 8
#define SPAWN_MAX_SPAWN_DATA SPAWN_MAX_SPAWNS*100
#define SPAWN_MAX_MESSAGE 120

#define SPAWN_IN_NULL(a) ((a)&spawn_flag_in_null)
#define SPAWN_CT_SIZE(a) ((a)>>8) & 0xff)
#define SPAWN_IN_SIZE(a) ((a)>>16) & 0xff)
#define SPAWN_OUT_SIZE(a) (a & 0xff)
#define SPAWN_T(a) ((spawn_t*)&spawn.st.spawn_data[spawn.st.spawn_idx[a]])

static const int spawn_flag_in_null = 1<<31;
static const int spawn_flag_local = 1<<30;
static const int spawn_flag_remote = 1<<29;
static const int spawn_flag_done = (1<<30) | (1<<29);

typedef void (*spawnable_fn_t) (const void*, void*, void*);
extern spawnable_fn_t spawn_functions[SPAWN_MAX_FUNCTIONS];
```

/* Data for each spawn
* optimized for local execution
* steal converts it to job_t
* in_t data is kept on the stack during fn call, until sync:
* - mem limited by #spawns/sync and #spawns_total
* - only after steals #spawns/sync is greater > max
*/

typedef struct {
  int fn;
  int flags; // in_null or in_t, ct_t, out_t (8 bit)
}
int sync;
const void *ct_p; // can be null
void *out_p; // can be null
char data[0]; // content of in_t (open ended)
}
spawn_t;

/* Data for each sync */
typedef struct {
    int pending, remote, isremote;
    int no_spawn;
} sync_t;

/* Spawn stack structures */
typedef struct {
    int cur_sync; // 1..no_sync (0 not used)
    int no_spawn; // next spot (not used)
    sync_t sync [SPAWN_MAX_SYNCS];
    char spawn_data [SPAWN_MAX_SPAWN_DATA];
    int spawn_idx [SPAWN_MAX_SPAWNS];
} spawn_stack_t;

extern spawn_stack_t spawn_st;

/* execute locally (C call):
 * depth > max
 */
extern int spawn_stack_cexec;

/* steal locally (create sync):
 * #spawns_total >= max or #spawns/sync > max
 */
static const int spawn_stack_sexec = 0;

/* initialize empty stack */
static inline void spawn_stack_init ()
{
    spawn_stack_cexec = 0; // spawn_stack_sexec = 0;
    spawn_st.cur_sync = 0;
    spawn_st.no_spawn = 0;
    spawn_st.spawn_idx[0] = 0; // first spawn pointer
    spawn_st.sync[0].no_spawn = 0;
    spawn_st.sync[0].remote = -1;
    spawn_st.sync[0].isremote = -1;
    spawn_st.sync[0].pending = -1;
}

/* push new empty sync
 * update: cexec
 */
static inline void spawn_stack_push_sync ()
{
    sync_t *p;
}
#ifdef DEB_STACK
if (spawn_stack_cexec)
    spawn_error("spawn_stack_push_sync: cexec");
if (spawn_st.cur_sync < 0)
    spawn_error("spawn_stack_push_sync: invalid cur_sync");
if (spawn_st.cur_sync >= SPAWN_MAX_SYNCS)
    spawn_error("spawn_stack_push_sync: too many syncs");
#endif

// save spawn pointer
spawn_st.sync[spawn_st.cur_sync].nospawn = spawn_st.nospawn;
// new sync
p = &spawn_st.sync[++spawn_st.cur_sync];
p->pending = p->remote = p->isremote = 0;
if (spawn_st.cur_sync >= SPAWNMAXSYNCS - 1)
    spawn_stack_cexec = 1;
}

/* pop sync
 * update; cexec
 * error: not empty
 */
static inline void spawn_stack_pop_sync()
{
    spawn_t *s;
    #ifdef DEB_STACK
    if (spawn_st.cur_sync <= 0)
        spawn_error("spawn_stack_pop_sync: no syncs");
    if (spawn_st.cur_sync >= SPAWN_MAX_SYNCS)
        spawn_error("spawn_stack_pop_sync: too many syncs");
    #endif
    s = SPAWN_T(spawn_st.no spawn - 1);
    #ifdef DEB_STACK
    if (spawn_st.no spawn > 0 &&
        ! (s->flags & spawn_flag_remote) &&
        s->sync == spawn_st.cur sync)
        spawn_error("spawn_stack_pop_sync: sync not empty");
    if (spawn_st.sync[spawn_st.cur sync].remote > 0)
        spawn_error("spawn_stack_pop_sync: remote not empty");
    #endif

    spawn_st.cur sync --;
    // restore spawn pointer
    spawn_st.no spawn = spawn_st.sync[spawn_st.cur sync].no spawn;
    if (spawn_st.cur sync < SPAWN_MAX_SYNCS - 1)
        spawn_stack_cexec = 0;
}

/* all spawns are finished? (last sync) */
static inline int spawn_stack_sync_is_done()
{
    return (spawn_st.sync[spawn_st.cur sync].pending == 0) &&
            (spawn_st.sync[spawn_st.cur sync].remote == 0);
}
*/ any remote steals ? (last sync) */
static inline int spawn_stack_sync_is_remote ()
{
    return spawn_st.sync[spawn_st.cur_sync].isremote;
}

/* print the stack in readable format */
void spawn_stack_print ();

/* allocate space for new spawn in last sync
 * update: pending, seexec
 * return spawn pointer
 * error: no syncs, #spawns>>, out of space
 */
static inline spawn_t* spawn_stack_save_spawn (int in_size)
{
    spawn_t *p;
    int tmpl, tmp2;
    #ifdef DEB_STACK
    if (spawn_st.cur_sync <= 0)
        spawn_error("spawn_stack_save_spawn: no syncs");
    if (spawn_st.no_spawn >= SPAWN_MAX SPAWNS - 1)
        spawn_error("spawn_stack_save_spawn: too many spawns");
    #endif

    tmpl = tmp2 = spawn_st.spawn_idx [spawn_st.no_spawn++];
    tmp2 += (sizeof (spawn_t) + in_size);
    spawn_st.spawn_idx [spawn_st.no_spawn] = tmp2;
    #ifdef DEB_STACK
    if (tmp2 >= SPAWN_MAX SPAWN_DATA)
        spawn_error("spawn_data out of space");
    #endif

    spawn_st.sync [spawn_st.cur_sync].pending ++;
    p = (spawn_t*) &spawn_st.spawn_data[tmp1];
    p->sync = spawn_st.cur_sync;
    return p;
}

/* deactivate last spawn; create new sync
 * update: pending, seexec
 * return NULL (empty sync) or spawn
 */
static inline spawn_t* spawn_stack_local_steal_push_sync ()
{
    spawn_t *p;
    if (spawn_st.sync[spawn_st.cur_sync].pending <= 0)
        return 0;
    #ifdef DEB_STACK
    if (spawn_st.no_spawn <= 0)
        spawn_error("spawn_stack_local_steal_push_sync: pending, but no spawns");
    #endif
    p = SPAWN_T (spawn_st.no_spawn - 1);
    #ifdef DEB_STACK
    #endif

    return 0;
}

/* any remote steals ? (last sync) */

if (p->flags & spawn_flag_done)
    spawn_error("spawn_stack_local_steal_push_sync: last spawn not valid");
if (p->sync != spawn_st.cur_sync)
    spawn_error("spawn_stack_local_steal_push_sync: last spawn not in current sync");
#endif

p->flags = p->flags | spawn_flag_local;
spawn_st.sync[spawn_st.cur_sync].pending --;
spawn_st.no_spawn --;
spawn_stack_push_sync();
spawn_st.no_spawn ++;

return p;
}

/* move up the stack: skip executed spawn & remote steals
 * pop sync */
static inline void spawn_stack_local_steal_pop_sync ()
{
    spawn_stack_pop_sync();
}

/* take first available spawn (keep 1 good spawn)
 * if none available, remote = spawn_no = 0
 * update: remote, seexec
 * return spawn pointer or NULL */
static inline spawn_t* spawn_stack_remote_steal (int *syncID)
{
    spawn_t *p;
    sync_t *s;
    int idx = 0;

    while (idx < spawn_st.no_spawn - 1) {
        p = SPAWN_T (idx);
        if (p->flags & spawn_flag_done)
            idx ++;
        else {
            // not the last spawn
            p->flags = p->flags | spawn_flag_remote;
            *syncID = p->sync;
            s = &spawn_st.sync[p->sync];
            s->remote ++;
            s->isremote = 1;
            s->pending --;
            return p;
        }
    }

    // no good spawns
    return 0;
}

/* update counter for stolen sync
 * update: remote, done */
static inline void spawn_stack_remote_done (int syncID)
{
    int tmp;
    #ifdef DEB_STACK
    if (syncID > spawn_st.cur_sync)
        spawn_error ("spawn_stack_done_remote: invalid syncID");
    #endif
    tmp = -- spawn_st.sync[syncID].remote;
    #ifdef DEB_STACK
    if (tmp < 0)
        spawn_error ("spawn_stack_done_remote: too many reports");
    #endif
}

#define SPAWN_STACK_SAVE_SPAWN(num, ct_t, ct_a, in_t, in_a, out_t, out_a) \
{ \ 
    spawn_t *sp = spawn_stack_save_spawn (sizeof(in_t)); \ 
    sp->fn = num; \ 
    sp->ct_p = (const void*) ct_a; \ 
    sp->out_p (void*) out_a; \ 
    \ 
    if (in_a) { \ 
        sp->flags = (sizeof(in_t)<<16) | (sizeof(ct_t)<<8) | sizeof(out_t); \ 
        *(in_t*)&sp->data = *in_a; \ 
    } else \ 
        sp->flags = spawn_flag_in_null | (sizeof(ct_t)<<8) | sizeof(out_t); \ 
}

#endif SPAWN_STACK_H
Implementation of tile phases in assembly:

.text

/*
Phase 1:
starts uintoff, ends uintoff & msg ACCEPT
reg 16..18,20..22 used but not saved (running as top)

- Handlers:
  handler m1 -> set RREJECT[i]
  handler m4 -> FAIL, freeze
  handler m2, m3 ->
    if RSTATUS != WAITING => FAIL, set error
    if m2 => RSTATUS = REJECT
    if m3 => RSTATUS = ACCEPT, read msg

- Main:
  RSTATUS = NO_MSG, RCOUNTER = RREJECT = 0
  cur_neighbor = 0

loop
  for every i, RREJECT[i] =>
    send M2 to neighbor[i], reset RREJECT[i]
  inoff
  if !(RREJECT and RSTATUS == ACCEPT) => j Phase2
  inon
  if RSTATUS == REJECT or RSTATUS == NO_MSG
    RCOUNTER ++
    if RCOUNTER == MAX =>
      RCOUNTER = 0
      cur_neighbor++ modulo no_neighbors
      RSTATUS = WAITING
      send M1 to cur_neighbor

end loop
*/

.global phase_1
.ent phase_1
phase_1:
  MTSRI PASS, 4100
  la $8,phase
  la $9,PH1
  sw $9,0($8)
  la RSTATUS, NO_MSG
  move RCOUNTER, $0
  move RREJECT, $0
  la $8, cur_neighbor
  sw $0, 0($8)
inoff

_loop_p1:
  # send M2s
la $16, neighbors # $16 = p += 4
la $8, no_neighbors
lw $17, 0($8) # $17 = no_neighbors
move $18, $0 # $18 = i
#MTSRI PASS, 4105
#msr PASS, RSTATUS
#msr PASS, RSTATUS
#msr PASS, RREJECT
#msr PASS, $17

_loop_m2:
la $10, 1
sll $10, $10, $18 # neigh bit
and $11, $10, RREJECT
beq $11, $0, _next_m2

# send M2 to i
lw $4, 0($16)
la $5, M2
MTSRI PASS, 4110
xor RREJECT, RREJECT, $10
jal send_msg

_next_m2:
addiu $16, $16, 4
addiu $18, $18, 1
bne $18, $17, _loop_m2

# Accept ?
uintoff
bnc RREJECT, $0, _skip_accept
la $8, ACCEPT
#MTSRI PASS, 4199
beq RSTATUS, $8, phase_2

_skip_accept:
uinton

# send M1 ?
la $8, REJECT
beq RSTATUS, $8, _next_counter
la $8, NO_MSG
beq RSTATUS, $8, _next_counter
j _loop_p1

_next_counter:
MTSRI PASS, 4120
addiu RSTATUS, RSTATUS, 1
la $8, MAX_COUNTER
bne RSTATUS, $8, _loop_p1

# send M1
move RSTATUS, $0
la $8, cur_neighbor
lw $9, 0($8) # $9 = cur_neighbor
la    $10, no_neighbors
lw    $11, 0($10)   # $11 = no_neighbors
addiu $9, $9, 1    # cur_neighbor ++

bne $9, $11, _no_adjust
move $9, $0

_no_adjust:
sw    $9, 0($8)     # write cur_neighbor
sll   $10, $9, 2
la    $8, neighbors
addu $10, $10, $8
lw    $4, 0($10)    # $4 = neighbor + cur_neighbor<<2
la    $5, M1
la    RSTATUS, WAITING
MTSRI PASS, 4150
jal   send_msg

j    _loop_p1

/*
 Phase 2:
 starts intoff - ends intoff

   - Handlers:
     handler m1 -> call spawn_interrupt_steal (msg)
     handler m2,m3 -> FAIL, freeze
     handler m4 -> call spawn_interrupt_report (msg)

   - Main:
     call spawn_run (msg)
*/

phase_2:
MTSRI PASS, 4200
la    $8, phase
la    $9, PH2
sw    $9, 0($8)

uintoff
la    $4, msg
la    $5, msg_len
jal   spawn_run    # return data in msg, msg_len
      # check intoff
uintoff

MTSRI PASS, 4299
j    phase_3

/*
 Phase 3:
 no uints - delayed until Phase 1
 no deadlock since parent always listens
*/
- Main:
  send msg/M₄ to cur_neighbor
  j Phase 1
*/

phase_3:
  MTSRI PASS, 4300
  la  $8, phase
  la  $9, PH3
  sw  $9, 0($8)

  la  $8, cur_neighbor
  lw  $9, 0($8)
  sll $9, $9, 2
  la  $10, neighbors
  addu $10, $10, $9
  lw  $4, 0($10)  # $₄ = neighbors[cur_neighbor]
  la  $5, M4
  MTSRI PASS, 4310
  MTSR PASS, $4
  jal  send_msg

  MTSRI PASS, 4399
  j  phase_1
.end phase_1
Appendix D

RAW Spawn Outputs

cagfarm-26 `\starsearch\module_tests\dynamic\s2results>tail -n 20 bin_tree-44-n26-C14-s10-c50

### PASSED: 00015004 00003a9c [x,y] = [0, 1]
### PASSED: 00016000 00003e80 [x,y] = [1, 1]
### PASSED: 00016004 00003e84 [x,y] = [1, 1]
[11: 00b65CC18]:
RESULTS: bin_tree(26)=67108864  clevel=14  cycles=191216684  xy=44  max_counter=50  spawn_max_syncs=10
### PASSED: 00017002 0000426A [x,y] = [1, 1]
### PASSED: 00017003 0000426B [x,y] = [1, 1]
### PASSED: 00001888 00000760 [x,y] = [1, 1]

DONE: 00000001 00000001 [x,y] = [1, 1]
[/\] *** interrupted [191223218]\inf]
stopped.
/*[469710566] 1*/
rm spawn_stack.s spawn_c2asm.s test_bin_tree.s spawn_asm.s
make[2]: Leaving directory `\am\cag-server\vol\volO\cag\home\ugradsOO\soviani\starsearch\module_tests\dynamic\spawn2'
Mon Jan 28 04:25:16 EST 2002

----------
cagfarm-26 `\starsearch\module_tests\dynamic\s2results>cat fib-12-n26-C15-s10-c50

[/\] welcome to beetle

[NUMBER  DEVICE NAME  RESET ROUTINE  CALC ROUTINE  PARAM]
[08405960  Serial Rom  dev_serial_rom_reset  dev_serial_rom_calc  08405960]
[0840d380  Print Service  dev_print_service_reset  dev_print_service_calc  0840d380]
[08407a78  DRAM  dev_dram_reset  dev_dram_calc  08407a78]
[0815a498  DRAM  dev_dram_reset  dev_dram_calc  0815a498]
[084387d8  DRAM  dev_dram_reset  dev_dram_calc  084387d8]
[084094e8  DRAM  dev_dram_reset  dev_dram_calc  084094e8]

[/\][0] 0*/
running...
[/\] [ serial rom : finished test.rbf --> static port 15 ]
[/\] *** interrupted [30898]\inf]
stopped.
running...
Interrupted tile 0, pc = 0x15c0
stopped.
running...

### PASSED: 00001100 0000044c [x,y] = [0, 0]
PASSED: 00001100 0000044c [x,y] = [1, 0]

PASSED: 00000000 00000000 [x,y] = [0, 0]

PASSED: 00000001 00000001 [x,y] = [1, 0]

PASSED: 00001110 00000456 [x,y] = [1, 0]

PASSED: 00001110 00000456 [x,y] = [0, 0]

PASSED: 00001200 000004b0 [x,y] = [0, 0]

PASSED: 00001201 000004b1 [x,y] = [0, 0]

PASSED: 00017000 00004268 [x,y] = [0, 0]

PASSED: 00017001 00004269 [x,y] = [0, 0]

PASSED: 00014000 000036b0 [x,y] = [0, 0]

PASSED: 00014005 000036b5 [x,y] = [0, 0]

PASSED: 00015000 00003a98 [x,y] = [1, 0]

PASSED: 00015004 00003a9c [x,y] = [1, 0]

PASSED: 00016000 00003e80 [x,y] = [0, 0]

PASSED: 00016004 00003e84 [x,y] = [0, 0]

PASSED: 00014000 000036b0 [x,y] = [0, 0]

PASSED: 00014005 000036b5 [x,y] = [0, 0]

PASSED: 00015000 00003a98 [x,y] = [1, 0]

PASSED: 00015004 00003a9c [x,y] = [1, 0]

PASSED: 00016000 00003e80 [x,y] = [0, 0]

PASSED: 00016004 00003e84 [x,y] = [0, 0]

PASSED: 00014000 000036b0 [x,y] = [0, 0]
cagfarm-26 `\~/starsearch\~/module_tests\~/dynamic\~/s2results` grep -h RESULTS *

RESULTS: bin_tree(20)=1048576 clevel=10 cycles=3382240 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=10 cycles=3197268 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=10 cycles=3406380 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=11 cycles=3203797 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=12 cycles=3414413 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=12 cycles=3461109 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=13 cycles=3767168 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=14 cycles=4402972 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=17 cycles=5063695 xy=44 max_counter=200 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=8 cycles=3383517 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=9 cycles=3272545 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=9 cycles=3406380 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=10 cycles=3652803 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=12 cycles=3767168 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: bin_tree(20)=1048576 clevel=13 cycles=67108864 clevel=14 cycles=191216684 xy=44 max_counter=50 spawn_max_syncs=10

RESULTS: fib(26)=196418 clevel=18 cycles=1024048 xy=33 max_counter=50 spawn_max_syncs=10
RESULTS: fib(26)=514229 clevel=15 cycles=2692737 xy=33 max_counter=50 spawn_max_syncs=13
RESULTS: fib(26)=82040 clevel=15 cycles=3983517 xy=33 max_counter=50 spawn_max_syncs=14
RESULTS: fib(26)=1346269 clevel=15 cycles=6228914 xy=33 max_counter=50 spawn_max_syncs=15
RESULTS: fib(26)=196418 clevel=15 cycles=686419 xy=44 max_counter=10 spawn_max_syncs=10
RESULTS: fib(26)=196418 clevel=15 cycles=750822 xy=44 max_counter=100 spawn_max_syncs=10
RESULTS: fib(26)=196418 clevel=15 cycles=774725 xy=44 max_counter=200 spawn_max_syncs=10
RESULTS: fib(26)=196418 clevel=15 cycles=691229 xy=44 max_counter=25 spawn_max_syncs=10
RESULTS: fib(26)=196418 clevel=15 cycles=726768 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: fib(26)=196418 clevel=15 cycles=754226 xy=44 max_counter=75 spawn_max_syncs=10
RESULTS: fib(26)=196418 clevel=15 cycles=709615 xy=44 max_counter=50 spawn_max_syncs=12
RESULTS: fib(26)=196418 clevel=15 cycles=744798 xy=44 max_counter=50 spawn_max_syncs=8
RESULTS: fib(26)=196418 clevel=18 cycles=805341 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: fib(26)=196418 clevel=9 cycles=804948 xy=44 max_counter=50 spawn_max_syncs=10
RESULTS: fib(26)=514229 clevel=15 cycles=7150652 xy=44 max_counter=50 spawn_max_syncs=13
RESULTS: fib(26)=82040 clevel=15 cycles=2679830 xy=44 max_counter=50 spawn_max_syncs=14
RESULTS: fib(26)=1346269 clevel=15 cycles=4325533 xy=44 max_counter=50 spawn_max_syncs=15
Appendix E

Pthreads Spawn Interface

ifndef SPAWN_H
#define SPAWN_H
#include <stdio.h>
#ifdef __cplusplus
extern "C" {
#endif

/* Spawn HEADER file */
* Any application using these functions needs to link in a spawn library specific to a system / implementation. The interface is not implementation dependent; the application can run on different platforms by linking different libraries.
*
* Terminology:
* spawn - spawned function call
* sync - all spawns at the same level (between syncs)
* thread - all syncs built by a thread entity, stack, etc.
*
* local execution - the spawn is executed right away, no state recorded
* save spawn - save the state of a spawn for later execution
* local steal - the spawn is executed from the local state (sync)
* remote steal - a spawn is taken from another thread, then executed
*/

/* INIT, DONE */
* Init the spawn system. More threads will be created; the main thread continues execution, while all the others will try to remote steal spawns.
* spawn_init_2D initializes the spawn system setting the size of the 2D array, the coordinates of the starting tile, the maximum spawn
* stack size, the maximum stack depth, and the size of the labels. No
* spawn functions can be used before calling spawn_init.
* Topology: xdim*xdim threads, communication is allowed only between
* adjacent threads i.e. \(|x1-x2| + |y1-y2| = 1\).
* spawn_done frees all resources used by the spawn system; no spawn
* functions can be used after this point.
*/

```c
void spawn_init_2D (int xdim, int ydim, int xinit, int yinit,
        int max_spawns, int max_depth,
        int cookie_len);
```

```c
void spawn_done (void);
```

```
/*
 * SYNC
 *
 * Adds a synchronization point, waiting for all spawns from the
 * lowest sync frame to finish. Possible actions:
 * - steal locally
 * - wait for all remote spawns to finish
 */

```c
void spawn_sync ();
```

```
/*
 * SPAWN
 *
 * spawn_define & spawn_callback_define are macros defining a
 * spawnable version of the function given as argument. A new function
 * symbol is defined - spawn_fn or spawn_callback_fn where fn is the
 * original name of the function.
 *
 * Forward declaration:
 * void fn_name (const ctt*, in_t*, out_t*);
 *
 * spawn_fn: write result to local frame; no access allowed to out_t
 * before sync().
 * => spawn_fn_name (const ct_t*, in_t *, out_t *)
 *
 * spawn_callback_fn: aggregate results to local frame. All function
 * calls associated with a thread fight for a single lock. The parent
 * thread and these functions don't share data. No access to agg_t
 * before sync().
 * => spawn_callback_fn_name (const ct_t *, in_t *, agg_t*
 * void (*) (out_t*, agg_t*)
 *
 * Possible actions of spawn (used by spawn_fn, spawn_callback_fn):
 * - execute locally
 * - save spawn
 * - steal locally (if the policy prefers earlier spawns)
 */

```c
typedef void (*spawnable_fn_t)(const void*, void*, void*):
```
void spawn (spawnable_fn_t fn,
    const void *ct, size_t cts, void *in, size_t ins,
    void *out, size_t outs);

void spawn_callback (spawnable_fn_t,
    const void* ct, size_t cts, void* in, size_t ins,
    void* out, size_t outs);

/*
 * CALLBACK FNs
 * input value, i/o value
 */
void add_int (const int*, int*);
void add_float (const float*, float*);
void add_double (const double*, double*);

/*
 * Access cookie string for current tile.
 * Get maximum length of cookie (set by spawn_init_X)
 */
char* spawn_cookie ();
int spawn_cookie_len ();

/*
 * Print a graphical representation of the tile graph.
 * For each tile its corresponding cookie is printed.
 */
void spawn_print_tiles (FILE* fp);

/* MACRO definitions */
#define spawn_define(fn_name, ct_t, in_t, out_t) \ 
    /* Forward declaration */ \ 
    void fn_name (const ct_t*, in_t *, out_t *);

inline void spawn_ ## fn_name (const ct_t *a, in_t *b, out_t *c) { \ 
    spawn ((spawnable_fn_t)&fn_name, \ 
        a, sizeof(ct_t), b, sizeof(in_t), c, sizeof(out_t)); \ 
}

#define spawn_callback_define(fn_name, ct_t, in_t, out_t, agg_t) \ 
    /* Forward declaration */ \ 
    void fn_name (const ct_t*, in_t *, out_t *, agg_t);

inline void spawn_callback_ ## fn_name (const ct_t* a, in_t* b, agg_t* c, \ 
    void (*d) (out_t*, agg_t*)) { \ 
    spawn_callback ((spawnable_fn_t)&fn_name, \ 
        a, sizeof(agg_t), b, sizeof(in_t), sizeof(out_t), \ 
    }
c, sizeof(agg_t), d); \\n
#endif
#endif

// SPAWN_H
Appendix F

Pthreads Spawn Examples

Definition of Fibonacci function:

```c
#include <stdio.h>
#include <stdlib.h>
#include "spawn.h"
#include "timing.h"
const int sleep_node = 10000, sleep_leaf = 100000;

/* Define an instrumented version of fib - spawn_fib */
spawn_define (fib, int, int, int);

/* Compute Fibonacci recursively */
void fib (const int* not_used, int* x, int* result)
{
    int xnew, res[2];

    if (*x <= 1)
    {
        sleep_uint64 (sleep_leaf); /* leaf "computation" */
        *result = 1;
    }
    else
    {
        sleep_uint64 (sleep_node); /* node "computation" */

        /* Recursion: 2 children are spawned */
        xnew = *x-1;
        spawn_fib (0, &xnew, &res[0]);
        xnew = *x-2;
        spawn_fib (0, &xnew, &res[1]);

        /* Synchronization: wait for both children to return */
        spawn_sync ();

        /* Use children's results only after sync */
        *result = res[0] + res[1];
    }
}
```
int main(int argc, char** argv)
{
    long long int t1, t2;
    int x, y, n, a=1, b=1, i;
    double runtime, work;
    int fib_res;

    /* Argument parsing */
    if (argc!=4) {
        fprintf (stderr, "%s xdim ydim n
",argv[0]); exit(1);
    }
    x = atoi(argv[1]); y = atoi(argv[2]); n = atoi(argv[3]);

    /* Computing Fibonacci iteratively */
    for (i=0; i<n-1; i++) {
        b = a+b; a = b-a;
    }
    printf("x=%d y=%d n=%d fib (%d)=%d
",x,y,n,n,b);

    /* Starting the spawn system */
    spawn_init_2D (x,y, x/2,y/2, 30,20, 16);

    /* Spawning fib */
    t1 = gettime_uint64();
    spawn_fib (0, &n, &fib_res);
    spawn_sync ();
    t2 = gettime_uint64 ();
    printf("finished: fib (%d) = %d
",n,fib_res);

    /* Stopping the spawn system */
    spawn_done ();

    /* Reporting */
    runtime = (double)(t2-t1)/1000.0;
    work = fib_res*(sleep_leaf/1000.0) + (fib_res-1)*(sleep_node/1000.0);
    printf("runtime: %g ideal rt: %g work: %g speedup: %g\n", runtime, work/(x*y), work, work/runtime);
    return 0;
}

Definition of binary heap traversal:

```c
#include <stdio.h>
#include <stdlib.h>
#include "spawn.h"
#include "timing.h"

int sleep_leaf = 10000, sleep_node = 1000;
bool print_on = false;
int *heap;

spawn_define (sum_heap, int, int, int);

void sum_heap (const int* heap_size, int *index, int *sum)
{
    if (print_on) {
        sprintf(spawn_cookie(), "%4d\n", *index);
        spawn_print_tiles (stdout);
        sprintf(spawn_cookie(), "X%4d
", *index);
    }
    if (*index > *heap_size) { 20
        /* Leaf node - fake computation by sleeping */
        *sum = 0;
        spawn_cookie("%4d\n", *index);
        sleep_uint64 (sleep_leaf);
    } else {
        /* Recursively compute sum of children
         * sum = f(2*b) + f(2*b+1) */
        int save_ch [2], newidx;
        sprintf(spawn_cookie(), "%4d\n", *index);
        sleep_uint64 (sleep_node);
        newidx = 2 * *index;
        spawn_sum_heap (heap_size, &newidx, &save_ch[0]);
        sprintf(spawn_cookie(), "%4d \n", *index);
        newidx++;
        spawn_sum_heap (heap_size, &newidx, &save_ch[1]);
        sprintf(spawn_cookie(), "%4dS\n", *index);
        spawn_sync();
        sprintf(spawn_cookie(), "%4d \n", *index);
        /* Compute partial sum */
        *sum = save_ch[0] + save_ch[1] + heap[*index-1];
        if (print_on)
            printf ("sum_heap(Xd) = %d\n", *index, *sum);
    }
} 40
```

int main(int argc, char** argv)
{
    long long int t1, t2, t3;
    int x, y, n, start;
    double runtime, work;
    int sum;

    /* Argument parsing */
    if (argc<6) {
        fprintf(stderr, "%s xdim ydim n sleep_leaf sleep_node [-print]\n", argv[0]);
        exit(1);
    }
    x = atoi(argv[1]); y = atoi(argv[2]); n = atoi(argv[3]);
    sleep_leaf = 1000*atoi(argv[4]); sleep_node = 1000*atoi(argv[5]);
    if (argc>6 && !strcmp(argv[6],"-print")) print_on = true;

    /* Initializing heap */
    heap = new int[n];
    for (int i=0; i<n; i++) heap[i] = 1;
    printf ("x=%d y=%d n=%d sleep_leaf=%d sleep_node=%d\n", x, y, n, sleep_leaf, sleep_node);

    /* Initialize spawn system */
    ti = gettime_uint64();
    spawn_init_2D (x, y/2, y/2, 30, 15, 5);
    t2 = gettime_uint64();
    printf("Done spawn_init_2D (%dus)\n",(int)(t2-t1));

    /* Start recursive process */
    start = 1;
    spawn_sum_heap (&n, &start, &sum);
    spawn_sync ();
    t3 = gettime_uint64();
    printf("Done computation (%dus): sum_heap(%d)=%d\n", (int)(t3-t2), n, sum);

    /* Stopping the spawn system */
    spawn_done();

    /* Reporting */
    runtime = (double)(t3-t2)/1000.0;
    work = (n+1)*(sleep_leaf/1000.0)+n*(sleep_node/1000.0);
    printf("runtime: %g ideal rt: %g work: %g speedup: %g\n", runtime, work/(x*y), work, work/runtime);
    return 0;
}

Appendix G

Pthreads Spawn Outputs

```
"/spawn/src>./fib 6 2 20
x=6 y=2 n=20 fib(20)=10946
finished: fib(20) = 10946
runtime: 124431 ideal rt: 100338
work: 1.20405e+06 speedup: 9.67641

"/spawn/src>./heap 4 1 15 100 100 -print
x=4 y=1 n=15 nodef=100000 snode=10000
Done spawn_init_2D (626us)
- - 1*
- - 2* -
- 3* 2. -
- 3* 4* -
- 3. 4. 5*
6* 7* 8* 5*
6* 7* 16* 5*
6. 7* 16* 5*
12* 7* 16* 5*
12* 7* 16. 5*
24* 7. 16. 5*
24. 14* 16. 5*
24. 28* 16. 5.
24. 28. 16. 10*
24. 28. 16. 20*
24. 28. 17* 20.
25. 28. 17. 20.
25. 29* 17. 20.
25. 29. 17. 21*
sun_heap(8) = 1
25. 29. 9* 21.
```

```
25. 29. 18* 21.
sun_heap(12) = 1
13* 29. 18. 21.
26* 29. 18. 21.
sun_heap(14) = 1
26. 15* 18. 21.
26. 30* 18. 21.
sun_heap(10) = 1
26. 30. 18. 11*
26. 30. 18. 22*
26. 30. 19* 22.
27* 30. 19. 22.
27. 31* 19. 22.
27. 31. 19. 23*

sun_heap(9) = 1
sun_heap(4) = 3
sun_heap(13) = 1
sun_heap(6) = 3
sun_heap(15) = 1
sun_heap(7) = 3
sun_heap(3) = 7
sun_heap(11) = 1
sun_heap(5) = 3
sun_heap(2) = 7
sun_heap(1) = 15
Done computation (591000us): sun_heap(15)=15
runtime: 591 ideal rt: 437.5
work: 1750 speedup: 2.99108
```

```
"/spawn/src>./heap 6 6 1023 100 100 -print
skipped ...
- - - - -
- - - - -
- - - 6. 3 7* -
- - - 4. -
- - - - -
- - - - -
```
skipped ... 5. 8. -
- - - - 28. -
- - 25. 26. 27. 29. -
- 13. 12. 14*
- 18* 9. 16. 10. 11.
- - - - -
- - - - -
12, 6, 3 wait on sync i.e. all outstanding
children are executed remotely
- - - -
- 51. 53. 55. 56. 57.
- 48. 50. 104* 54. 58. 59.
- 49. 128 65 35 60. 62.
- 36. 37. 38. 64* 40. 44*
- - - 35* 43* 47* 10. 8. 9.
skipped ...
824. 816. 1696* 1760* 896. 912.*
768. 1600* 1664* 1728* 928. 944.*
744. 128 65 35 960* 992*
576* 592* 608* 1024* 640* 352.*
584* 600* 624* 544* 672* 736*
588* 604* 632* 560* 688* 752*
skipped ...
"\spawn\src>.\heap 3 3 63 100 10 -print
x=3 y=3 n=63 sleaf=100000 snode=10000
Done spawn_init_2D (1068us)
- - - -
- 1* -
- - -
- 2* -
- - -
- 3* -
- 2. -
- - -
- 3. -
- 4* -
- - -
- 5* -
- 4. -
- - -
- 6. 7*
- 4. -
- - -
- 6. 7*
- 4. -
- - -
- 6. 7*
- 5* 4. -
- - -
- 12* 7.
- 5. 8 -
- - -
- 13* 12. 7.
sum_heap(38) = 1
38  49  57.
41  33  37.
45  93  39

skipped ...

sum_heap(60) = 1
60. 102. 118.
111. 2  61.
43  63. 126.

sum_heap(62) = 1
110. 102. 118.
111. 2  61.
126. 127* 62

sum_heap(55) = 1

sum_heap(27) = 3
sum_heap(13) = 7
sum_heap(63) = 1
sum_heap(31) = 3

sum_heap(51) = 1
sum_heap(25) = 3
sum_heap(12) = 7
sum_heap(6) = 15
sum_heap(59) = 1
sum_heap(59) = 3
sum_heap(14) = 7
sum_heap(61) = 1
sum_heap(30) = 3
sum_heap(30) = 3
sum_heap(15) = 7
sum_heap(7) = 15
sum_heap(3) = 31
sum_heap(1) = 63

Done computation (591000us): sum_heap(15)=15
runtime: 591 ideal rt: 437.5
work: 1750 speedup: 2.96108
Bibliography


