Optimizing Memory Accesses for the Architecture Exploration System (ARIES)

by

Eric Y. Mui

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

June 2000

Author_____
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by_____
Professor Srinivas Devadas
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Optimizing Memory Accesses for the
Architecture Exploration System (ARIES)
by
Eric Mui

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2000

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

Abstract

The demand for devices with embedded systems is increasing rapidly. Short design cycles and increasing complexity of the systems dictate that most of the functionality of the device be implemented in software, and cost/performance/power requirements argue for processors tuned to the particular application performed. In order to effectively explore the hardware/software design space, a retargetable compiler is necessary. The AVIV compiler within the Architecture Exploration System is such a compiler. For architectures with limited addressing modes and support for auto-increment and auto-decrement instructions, optimizing the layout in memory of program variables can reduce the number of generated assembly instructions devoted to address pointer arithmetic or loading address registers. Reducing the overall size of the generated code has the effect of reducing the cost of the embedded system. This thesis presents a design for integrating this optimization with the AVIV retargetable compiler.

Thesis Supervisor: Srinivas Devadas
Title: Professor

Acknowledgements

I have had a wonderful time here at MIT, during my undergraduate and graduate years. I am indebted to both my friends and the faculty who have helped me grow both personally and intellectually.

I would like to thank my thesis advisor, Professor Srinivas Devadas, for his patience and guidance during my research. I could always drop in anytime and discuss problems I encountered, and leave feeling both encouraged and having a renewed sense of purpose. His friendly, easy-going manner made my first real research experience a pleasure.

I would also like to thank George Hadjiyiannis for his patience and helpfulness when I first joined the Computer-Aided Automation group. His careful, detailed introduction to ISDL and the ARIES project was an immense help to me in understanding not only the system and its details, but the bigger picture of the purpose behind ARIES as well. His advice, and sense of humor, also added to the enjoyment of my time in CAA.

I am much in debt to my friend Xiaobo Li who encouraged, cajoled, and otherwise kept me on the right path to finishing this thesis. I am forever grateful for her years of friendship, and look forward to many more.

Finally, I wish to thank my family, whose support for me in whatever I did never faltered. It truly means a great deal to me, and I continue to cherish it.

Contents

List of Figures and Tables

Chapter 1

Introduction

The demand for consumer electronics and telecommunications products has increased dramatically in the past few years. Cellular phones, portable multimedia devices, and personal digital assistants are some examples of these complex systems. Increased functionality, lower cost and power consumption, reduced size, and being first-to-market often are deciding factors in the success of these products. In order to support these goals, designers are building entire systems on a single die, with embedded core processors becoming vitally important to manage the complexity of adding functionality, reducing product cycle times, and still retaining flexibility in the face of evolving standards [17].

Embedded systems, specialized for a single application instead of general-purpose computation, increasingly have both hardware and its controlling software integrated on a single chip. This allows the design to satisfy requirements of low cost and power consumption, and reduced size. Time-to-market requirements place greater burdens on system designers for shorter design cycles, leading to a greater portion of the system functionality being implemented in software. As the density of transistors increases, the hardware portion of a system requires less chip area relative to the software, which

resides in the program ROM. Reducing the code size leads to a reduction in the area required by the program ROM, reducing overall chip area and thus IC cost. Real-time performance requirements of most embedded systems are also a driving force for producing high-quality, compact code. A *hardware-software co-design* methodology, in which both hardware and software are designed together to improve the final implementation of the system, would afford the most benefit in reducing costs and increasing performance.

The drive to produce more compact, high performance code for embedded systems is hampered by current high-level language compilers. Current compilers for fixed-point DSPs generate code that is unsatisfactory with respect to code size and performance. Thus most DSP software is handwritten in assembly, a time-consuming task. Choosing different target architectures would require this handwritten code to be rewritten. Applications running on embedded systems are also becoming standardized, calling on an applications program interface (API). Embedded processors need only implement the functions available through the API, giving designers more freedom to choose processor architectures. The lack of retargetable compilers hampers efforts to develop architectures tuned to the specific application.

An ideal solution would allow system designers to compile application code to a minimal size on a variety of architectures to explore which is the best for the given application. The compiler in the Architecture Exploration System (ARIES) [1] does this, using the application source code and a description of the target architecture as inputs.

1.1 Project ARIES

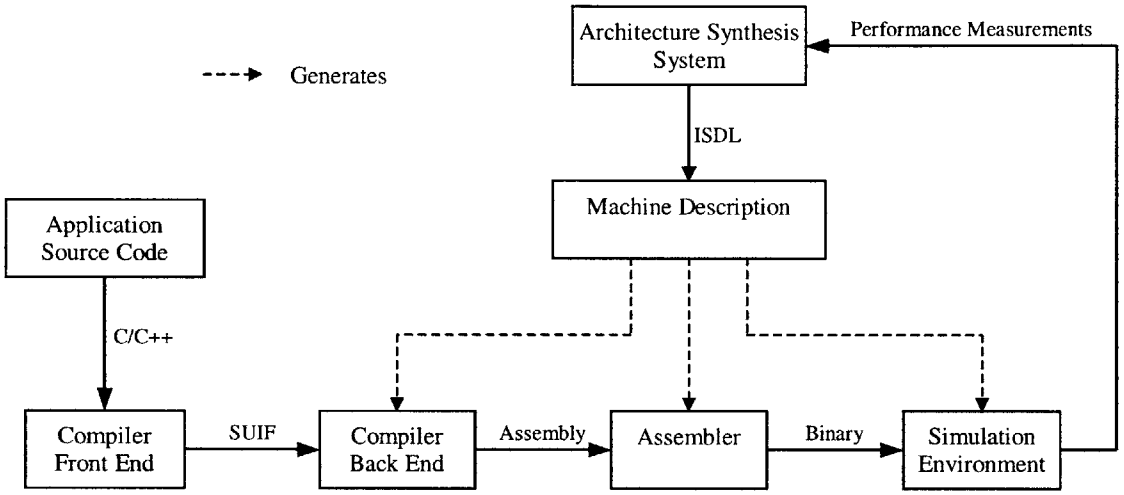

Figure 1.1: Architecture exploration using the Architecture Exploration System (ARIES).
Figure taken from [1].

The Architecture Exploration System, shown in Figure 1, is a framework for pursuing a hardware-software co-design methodology. ARIES receives as inputs the application source code in C or C++ and a description of the target architecture, which is written in ISDL, or Instruction Set Description Language [1]. ARIES parses the ISDL description file for the architectural details of the target machine, including the amount and types of memory, the format of the instruction word, the instructions defined on the architecture, and the constraints on the interaction of different parts of the hardware. After parsing the ISDL description, tool generators automatically create a compiler back-end, assembler, disassembler, and an instruction-level simulator. The application code passes through the compiler front-end, and a machine-independent representation is created in the Stanford University Intermediate Format (SUIF) [4]. The automatically generated compiler back-end and assembler convert the SUIF code into binary code. In

the simulation environment the performance of the compiled code is measured, and based on the results the ISDL description can be modified to improve performance. Then the cycle of compile, test, and modification begins again, until suitable performance is achieved. The compiler back-end and assembler allows the compiler to be retargetable without requiring the developer to write a new back-end and assembler for each of the new architectures evaluated.

1.2 Reducing Code Size by Optimizing Memory Layout

Memory addressing modes in DSPs are inferior to general-purpose processors, though the lack in functionality is made up for in faster performance in computation [14]. Data is addressed through register pointers to memory, and often there is a set of "address registers" dedicated to this purpose. DSPs usually support auto-increment or auto-decrement load and store instructions, which in addition to loading or storing data values adjust the current pointer by $\pm k$ memory locations in the same operation, where k is some small integer. The usage of the terms "auto-increment" and "auto-decrement" in this thesis is meant to imply post-modify operation, in which the value of the address pointer is modified after the load or store operation. When the target address of the next load or store is greater than distance k from any pointers, then an instruction is needed to set the value of a pointer to the needed address. Optimizing the arrangement of variables in the program, so that variables often accessed together were adjacent in memory, would minimize the number of instructions needed for setting the value of address pointers. This in turn leads to some reduction in the overall code size of the program ROM, thereby achieving some reduction in cost.

1.3 Thesis Roadmap

This thesis presents a method for adding capability to the ARIES compiler to support optimizing generated code for architectures that have limited addressing modes, and specifically architectures with auto-increment/decrement instructions. Chapter 2 discusses previous research related to optimizing memory accesses on architectures with limited addressing modes. Chapter 3 presents a method for determining the optimized variable layout and generating code using such a storage layout. Chapter 4 presents a sample of results from generating code with and without optimization for comparison. Chapter 5 concludes this thesis and discusses directions for further work.

Chapter 2

Related Work

There are a number of papers in the literature that focus on the problem of optimizing variable layout for fixed-point DSPs with limited addressing modes to reduce code size. Most of these papers target architectures such as the Texas Instruments TMS320C25, which has support for auto-increment/decrement implicit memory accesses.

2.1 Optimizing Stack Frame Accesses

Bartley in [5] presented an approach for optimizing the layout of local variables stored in the stack frame of processors that do not have register-plus-offset addressing. Especially for architectures with auto-increment/decrement instructions, optimizing the variable layout had an impact on the size and speed of the generated code.

Without register-plus-offset addressing, variables in the stack frame would have to be accessed through a register pointing to memory, which Bartley referred to as a roving pointer (RP). To optimize the variable layout for use with auto-increment/ decrement instructions, Bartley created an undirected weighted graph $G(V, E)$. The vertices V represented the variables of the program and the edges E were weighted by the expected benefit of having the pair of variables allocated contiguously in memory. If all

pairwise permutations of variables are considered, the graph G must be complete. In this case an edge E may have zero weight, which indicates there is no benefit from having the two corresponding variables adjacent in memory, but is allowed nonetheless. Finding an optimal ordering for the variables is equivalent to finding a Hamiltonian path in G that maximizes the sum of the weights along the path. Adjacent vertices along the Hamiltonian path correspond to variables that should be allocated adjacent to each other in memory.

Determining the Hamiltonian path that maximizes the sum of the weights along its edges for G is NP-complete [5], so a heuristic solution was proposed. The solution involves incrementally removing edges until no cycles remain and all vertices have no more than two neighbors. The edge with the highest weight is considered for removal or retention in G at each step. Since the weight of each edge is directly related to the number of times its vertices are accessed sequentially, retaining edges with higher weight is preferred, unless the edge causes a cycle with other edges that are already in the solution. Once a vertex has two incident edges to be retained, any remaining incident edges are removed from G. Determining a layout for the variables simply involved following the path chosen by the heuristic solution and assigning consecutive vertices to consecutive stack locations.

2.2 Simple Offset Assignment/General Offset Assignment

Liao, in his Ph.D. thesis [8], presented a formulation of the same problem Bartley studied, calling it the simple offset assignment problem (SOA). SOA optimizes the layout of automatic variables of a procedure such that the number of instructions generated to

access those variables is minimized. The solution to SOA involves creating a graph similar to the one Bartley used, called an *access graph*, though with zero-weight edges were eliminated. Once this "access" graph is created, then Liao formulates SOA as a graph-covering problem, which he termed *maximum weight path covering* (MWPC). Solving MWPC involved determining a *disjoint path cover* of the graph that minimized the sum of uncovered edge weights.

The disjoint path cover was determined by selecting edges in the access graph with the heaviest weights. A separate, empty graph that represented the disjoint path cover was created. Sorting the edges in the access graph by weight, the heaviest one was selected. If the addition of the selected edge caused a cycle in the disjoint path cover graph, or caused any vertex in the cover graph to have more than two incident edges, the edge was discarded. Otherwise the selected edge was added to the cover graph. In either case the edge was removed from the access graph, and the process repeated. Eventually all the edges in the access graph would be processed. The edges in the disjoint path cover might not form a single path, hence the disjoint nature of the graph covering. Creating a variable layout involved following each disjoint path and assigning adjacent memory locations to adjacent vertices in the path. Address register load instructions would be needed when an address register jumped from one disjoint path to another.

Liao extended SOA to the general offset assignment problem (GOA) in [8], which optimized the layout of variables when there was more than one address register available. The solution to GOA required finding a partition of the variables that allowed the disjoint path cover to minimize the uncovered edges and also to minimize the setup costs of using the multiple address registers. The quality of the partition, in terms of

choosing the variables that would lead to the minimal cost, greatly influences the quality of the layout assignment generated by GOA.

Leupers and Marwedel presented a more optimized method of choosing partitions for GOA in [10]. For solving GOA with k available address registers, l partitions were created, $l \leq k$. The edges in the access graph were sorted in decreasing order of weight, and the l heaviest disjoint edges were selected, one edge to initialize each partition $V_1 \ldots V_l$ with its two vertices. There could be at most k partitions, though there could be fewer if there were insufficient numbers of disjoint edges. For each of the remaining vertices in the access graph, the vertex was temporarily added to each partition, and the cost of adding that vertex for each partition was evaluated. The vertex would be permanently added to the partition that had the smallest increase in cost. Once all vertices were assigned, the final variable layout was generated as the concatenation of the variable layouts of each partition.

In [11] Liao illustrated an extension of SOA and GOA to utilize instructions that could auto-increment/decrement an address register by $\pm l$, instead of by only unit increments. Termed the l-simple offset assignment (l-SOA) problem and (l,k)-generalized offset assignment ((l,k)-GOA) problem, using instructions having increment/decrement value of l allows edges that could not previously be included in the disjoint path cover, due to cycle or incident edge constraints, have effectively zero cost. To identify these costless edges, however, the access graph had to be complete graphs instead of graphs with only positive weight edges. For a complete graph $G(V,E)$ and a cover C of G, a subgraph of G with (l+1) vertices that was a subpath of C with length l was called an *induced* (l+1)*-clique*. A zero-weight edge in C could allow a positive-weight edge not in

C to be in an induced (l+1)-clique. Since the vertices of the positive-weight edge were within a distance of l of each other, the positive weight-edge effectively has no cost in terms of extra instructions needed to load an address register with an address. Such an edge is said to be *induced* by C. The l-SOA problem minimized the sum of the weights for edges not covered by either the cover C or any induced edges, called the *induced cost.* (l,k)-GOA finds a partitioning of variables such that the sum of the induced cost for each partition plus setup cost of address registers used is minimized.

2.3 IALOMA/ALOMA-CCG

In work presented in [12], [13], [14], and [15] a different approach for creating the access graph and determining the layout of variables with auto-increment/decrement instructions was used. As before, the vertices of the access graph were the variables to be laid out in memory. However, the weight of each edge (v_i,v_j) represented the "distance" between memory accesses for variables v_i and v_j. The "distance" between memory accesses is the number of instructions between these accesses inclusive, so loading two variables from memory one right after another has a distance of 1, and one instruction between the two loads leads to a distance of 2. There may be more than one edge between two vertices, and each edge may be of different weight, as each edge represents each occurrence of the memory access for one vertex following the memory access of the other vertex of the edge.

Certain aspects of the addressing modes for the architecture assumed by [12], [13], [14], and [15] bear mentioning. First, the assumption is made that memory is only indirectly addressed by one or more address registers. Second, during one instruction

cycle an address register can be incremented or decremented by $\pm k$ (where k is a small positive integer, typically less than 8) in parallel with an arithmetic operation or data movement from one memory to another. Third, an address register can be updated by ±k during any instruction cycle, regardless if the register is used for an actual memory reference during that cycle. Finally, loading an address register with an address costs one instruction cycle.

2.3.1 IALOMA

The problem of finding an efficient memory address allocation is called the "memory allocation problem", and the proposed solution following an algorithm called IALOMA (Improved Address LOad Minimization Algorithm). The memory allocation problem assumes that only one address register is available. Since the architecture supports modifying an address register without requiring an actual memory access, between two memory accesses there may be some number of instruction cycles available to modify the address register. Thus there is a possibility to point the address register to the desired memory address without the need to explicitly load an address into the register, even if the addresses are further than ±1 apart. In the access graph, if an edge has a weight greater than the number of vertices in the access graph, then it is always possible to adjust the address register to point from one vertex to the other without the need for an explicit load. Thus, any such edge is considered redundant and removed.

From the access graph, a number that represents the benefit of removing an edge from the graph is assigned to each edge. This "benefit" takes into account the branching factor of the vertices of the edge, whether the edge is part of a cycle, and the distance

between the vertices for this edge. According to [12], the benefit of an edge is an estimate of how much the access graph is linearized by removing that edge, and at the same time favoring edges that have a greater distance for removal.

After giving each edge in the access graph a "benefit" number, determining an assignment for the memory layout of variables is performed by iteratively removing edges of higher benefit, and recalculating the "benefit" for each edge, until a linear graph has been derived from the access graph. Since removal of an edge affects the branching factor for some vertices of the graph, as well as the presence of cycles, the relative benefit of removing the remaining edges must be recalculated. Once the access graph has been reduced to a linear graph, start at one end of the linear graph and number sequentially to derive the memory layout.

IALOMA is in some ways similar to the approach used to solve SOA. Whereas in IALOMA each edge is given a weight that describes the "benefit" of removing that edge, in SOA each edge is given a weight that describes the cost of removing that edge. Since highest weight edges are preferentially removed in IALOMA and preferentially retained in SOA, then both solution methods attempt to keep high cost edges in the access graph. Additionally, SOA and IALOMA attempt to find linear paths through an access graph.

The problem of utilizing more than one address register to improve memory accesses is called the "AR assignment problem". Solving the AR assignment problem involves determining a partition of the variables into groups, with one address register for each group. Since the variables have been partitioned, the sequence of accesses for each group only includes those variables in the partition, so each group has its own access graph. The distance between members of a group has changed as well, and they need to

be determined. After the access graph for a group has been determined, IALOMA is applied to each group to determine a memory layouts for each. This strategy is very similar to the one used to solve GOA.

2.3.2 ALOMA-CCG

ALOMA-CCG (Address LOad Minimization Algorithm with CCG-(k+1)) [14] is a variation on IALOMA that considers DSPs with the ability to update an address register within the range $\pm k$, rather than being limited to ± 1. The ALOMA-CCG approach extracts cliques of size (k+1) where k is the limit of the update range for the auto-increment/decrement instructions. A clique of size (k+1) can be covered with an increment or decrement of k without the need to load an address into an address register. Such a clique was called a Chained Clique Graph-(k+1), or CCG-(k+1). Within a clique a relative ordering can be found for the variables without requiring a load to an address register. When more than one CCG-(k+1) share a common edge, it is possible to connect them together to form a quasi-CCG-(k+1). As with the CCG-(k+1), the quasi-CCG-(k+1) can determine an ordering of variables without the need for a load of the address register. Adding a CCG-(k+1) to a quasi-CCG-(k+1) leads to another quasi-CCG-(k+1), so for a given access graph, it is desirable to extract the largest quasi-CCG-(k+1) to minimize the number of address register loads needed.

The architectural assumption that an address register can be modified without being used in a memory reference becomes important in determining the memory allocation. Since we can adjust the address of the address register during any instruction cycle, a distance greater than k between two sequentially accessed variables can be

covered by incrementing or decrementing while performing other instructions, provided there are enough intervening instruction cycles between the two sequential memory accesses. However, if the two variables were to be addressed in consecutive instruction cycles and the distance between them is greater than k, then an address register load would be required.

2.4 Optimizing Array Element Accesses

In the work by Araujo, Sudarsanam, and Malik [6], memory accesses for array elements in a for-loop were optimized using address registers. First, the array elements were numbered in the order that they were accessed. These numbered array elements became the vertices in a directed graph called an *indexing graph* (IG). There was a directed edge between two vertices in the IG if the *indexing distance* between them is at most the step size of the auto-increment/decrement. The step size is the amount by which an address register can be incremented or decremented using one instruction. The *indexing distance* is a function of the indices of the two array elements, and is used to determine if an address register can be used for both array accesses. To make the problem more tractable, edges that reached over multiple iterations were removed from the IG graph. Assigning address registers for the array accesses involved minimizing the number of disjoint paths needed to cover the directed graph. This strategy of using disjoint paths to cover the IG is also similar to the approach used by Liao to solve SOA, though in SOA the access graph did not have directed edges.

Chapter 3

Memory Layout Optimization and Code Generation

3.1 Roadmap for Memory Layout Optimization and Code Generation

Optimizing the layout of variables in memory and generating assembly code for an input C program occurs on the procedure level. The approach used to perform memory layout optimization is based on the method presented in [8] for solving GOA with a given number of address registers, and limiting updates of the address registers to ± 1. The steps involved in optimizing the memory layout for an input C program, and emitting assembly code for that program, are as follows:

1. Use SUIF compiler (Section 3.2) to process and create an intermediate form representation of the C program.
2. Examine the intermediate form of each procedure in the C program to determine the access sequence of that procedure, and build an access graph (Section 3.3.1) that also captures information of how access sequence may change due to breaks in control flow.
3. Solve GOA problem on the graph, and determine the optimized layout for the variables in the procedure. (Section 3.3.2)

4. Using the optimized layout and intermediate form, generate assembly code for each procedure, inserting load, store, and address register load instructions as needed. (Section 3.4)

Figure 3.1 illustrates the overall steps involved in optimizing layout and generating assembly code.

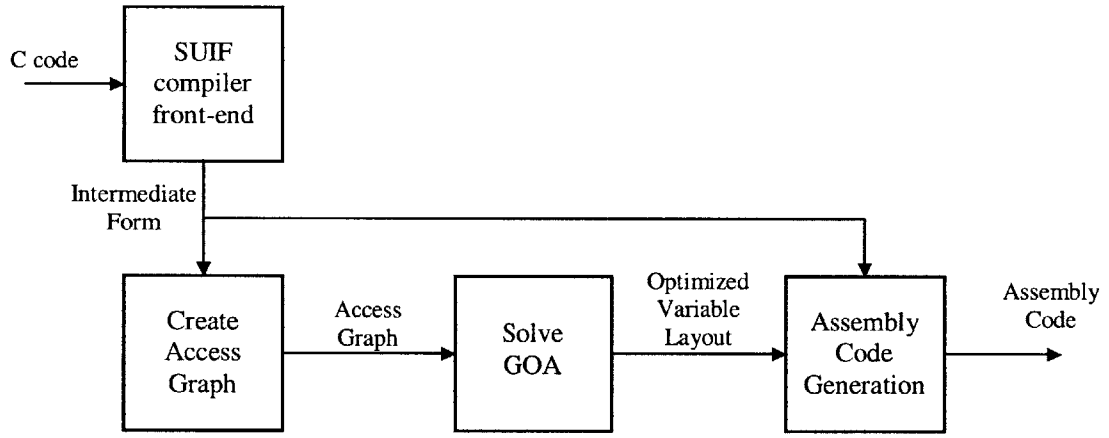

Figure 3.1: Steps in optimizing variable layout and code generation.

3.2 The SUIF Compiler

The Stanford University Intermediate Format (SUIF) compiler [4] forms the front-end of the AVIV retargetable compiler [3][16]. In order to be able to incorporate optimizing memory layout into AVIV, the same front end was used.

The SUIF compiler is used to generate an intermediate representation (IR) of the input C program. One feature of the IR that lent itself for use in optimizing memory layout was that control flow statements could be kept in a high-level representation that allows control flow information to be preserved, while converting other types of instructions to a low-level representation, suitable for generating assembly code. The

body of a procedure can be considered a list of instructions, or in the case of branches in control flow structures that contain separate lists of instructions for each branch of control.

Mostly there are two types of elements that compose a procedure body in the IR. The majority of a procedure is composed of simple three-operand instructions found in most RISC-like architectures, such as arithmetic operations, logical operators, and comparison operations. There are also branch structures, such as an IF or WHILE statement. The branch structures maintain separate lists of instructions for the conditional test and the body of the structure, of which there may be more than one, depending on the type of branch. An IF structure, for example, has a header section for the condition test, a then-part, and an else-part. The instructions in the header section evaluate to a Boolean value, and the instructions within the lists for the then-part and else-part correspond to the statements in the C program that make up those parts of the if-statement.

3.3 Generating a Memory Layout Assignment

Generating an assignment for layout of program variables in memory involves deriving the access graph for each procedure under analysis, and solving GOA for the derived access graph.

3.3.1 Creating the Access Graph

The access graph is derived from the *access sequence* for a procedure. The access sequence is determined by recording the order in which variables were accessed to

perform an operation. For the operation $a = b$ op c, the access sequence would be $b\ c\ a$. Figure 3.2a shows a series of simple operations, and the resulting access sequence.

For branch structures like the IF statement, since there is more than one direction the control flow may follow, the access sequence of the separate directions of flow are kept separate from each other, and from the access sequence of the previous instructions or following instructions. However, the access sequence of the conditional test can be considered part of the access sequence of the previous instructions, since these instructions must be performed before a branch decision can be made. Figure 3.2b illustrates how the access sequence is determined for an IF branch structure.

The overall access sequence for a procedure is a concatenation of all the separate accesses sequences, as shown in Figure 3.2c. Where the control flow may diverge, a new access sequence is started for each branch. Where divergent flows of control meet, a new access sequence is also started. The contiguous access sequences correspond to basic blocks within the procedure. The arrows indicate the directions that control flow can follow.

The access graph summarizes the pattern in which variables are accessed. As discussed in [8], each vertex v in access graph G corresponds to a unique variable in the access sequence. An edge $e = (u,v)$ exists between vertices u and v if they are adjacent to each other in the access sequence, and has weight $w(e)$ which represents the number of times u and v are adjacent. There is no difference in the relative ordering of u and v, so edge (u,v) and edge (v,u) are considered to be the same edge. In creating G for an access sequence with control flow, an edge is added between the last variable of a contiguous access sequence and the first variable of any preceding or following contiguous access

sequences. Figure 3.3 shows the access graph derived from the access sequence shown in Figure 3.2c.

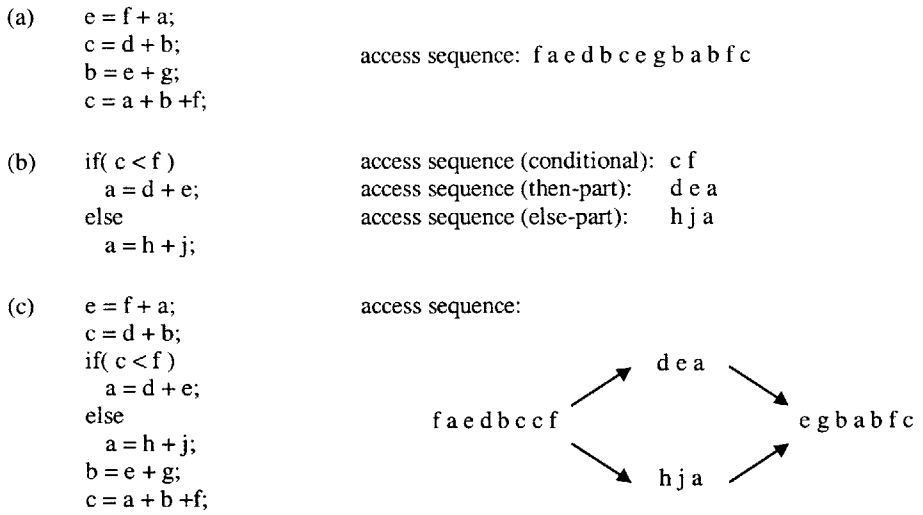

Figure 3.2: Examples of access sequences.
(a) The access sequence for a series of simple instructions is straightforward.
(b) The access sequence for an IF branch structure is actually composed of three separate access sequences.
(c) The access sequence for a combination of simple instructions with an IF branch structure.

3.3.2 Solving General Offset Assignment

Once the access graph has been created, the next step in optimizing the variable memory layout is solving GOA on the access graph. Solving GOA, as presented in [8], is made up of a number of steps:

1. First solve SOA on the given access graph using one address register.
2. Set n to the number of address registers used plus one, and partition the variables in a way that would yield the most benefit for using n registers address registers.
3. Determine a new access sequences for each partition of the variables, based on the original access sequence, and create new access graphs for each new access sequence.
4. Solve SOA on each new access graph.

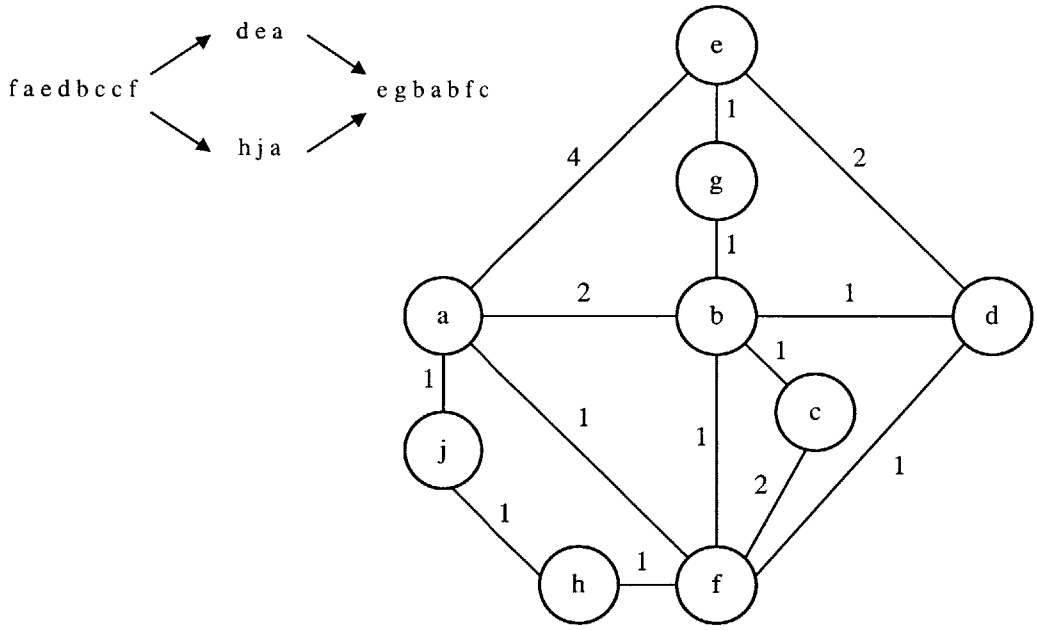

Figure 3.3: Access sequence and the corresponding access graph.

5. Sum the costs of covering each new access graph and the cost of initializing n address registers. Compare this cost to the previous cost of using n-1 address registers. If the cost for n-l address registers is higher, partitioning makes sense so the new memory layout assignment is chosen, and steps 2-5 are repeated. If the cost of using n address registers is higher, nothing can be gained from additional address registers, and the memory layout assignment for n-1 address registers is chosen.

The final memory layout assignment is a concatenation of the separate assignments generated by SOA for each address register used, since ultimately all the variables will be allocated in a linear order in memory.

As described in [8], the cost associated with finding a disjoint path cover for an access graph in SOA is the sum of the weights of uncovered edges in the graph and the cost of initializing an address register. This cost metric reflects the additional instructions needed to set the address register to the correct address for memory references of variables not adjacent to each other. Only one address register is used in SOA. Figure 3.4 shows a disjoint path cover, using one address register, of the access graph in Figure 3.3, and the associated cost of the cover.

Finding a "good" partition of the program variables has a large impact on the effectiveness of GOA. Figure 3.5 illustrates a partition of the access sequence of Figure 3.3. There is no one method for determining how to separate variables into groups in all cases such that the cost of covering the derived access graphs is minimized. A method that works well for one C program might not work well for another, depending on the access patterns of the variables. The method of partitioning used in this thesis is based on the observation in [8] that vertices with many incident edges correspond to variables that are accessed frequently. Having a separate address register pointing to those variables may provide the most benefit in reducing the number of address register load instructions. However, deciding how to partition the variables into separate partitions was not clear. An approach such as the one suggested in [10] seemed unnecessary, so partitions were decided on whether a given vertex had greater than the average number of incident edges for the access graph. The average was simply the arithmetic mean of the number of incident edges to each vertex.

After solving GOA on the access graph of the procedure under analysis, the resulting assignment is recorded into a table for code generation. There is a separate table

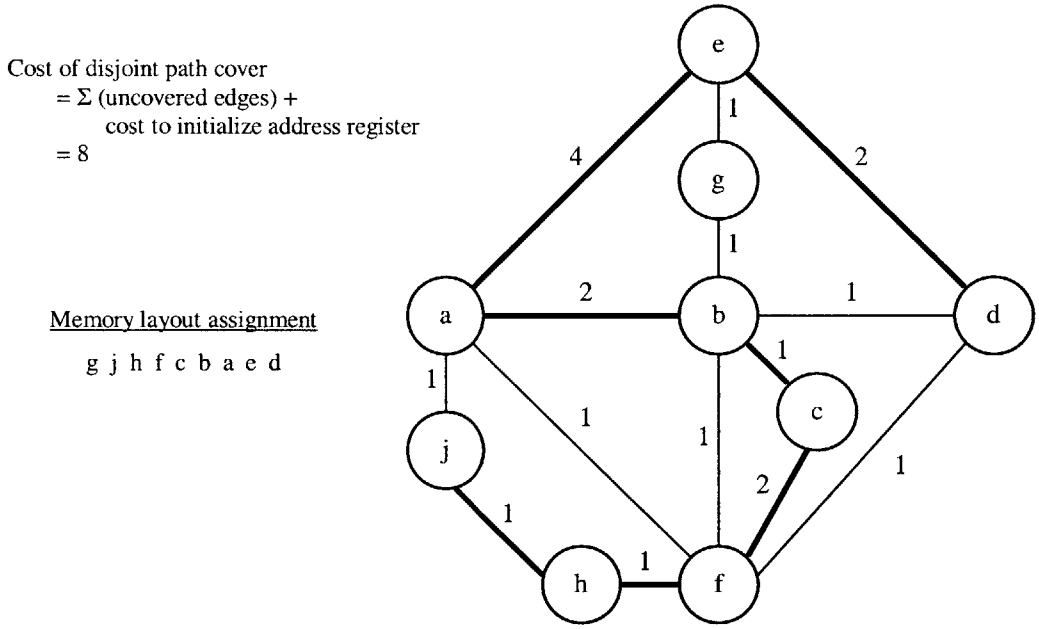

Figure 3.4: Disjoint path cover, associated cost of the cover, and resulting memory layout assignment. Heavier lines indicate edges that are included in the disjoint path cover, and lighter lines indicate edges that are excluded from the cover. As indicated, the total cost includes the cost to initialize the value of the address register, in this case assumed to have a cost of 1.

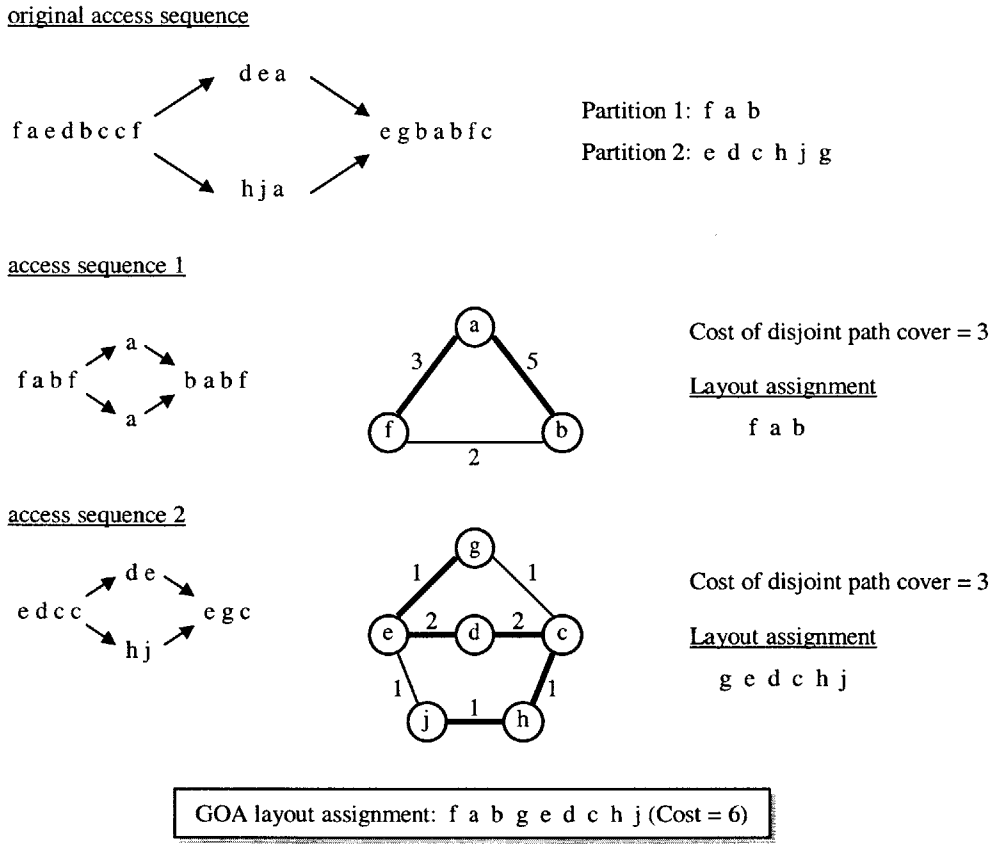

Figure 3.5: Partition of variables (with two address registers) leads to reduction in overall cost.

for each address register in use, since each address register has its own assignment generated by SOA. The relative position of each variable within the overall GOA assignment is recorded as well. These relative positions are used during code generation to determine whether consecutive memory references will require address pointer arithmetic, an explicit load of an address register, or an auto-increment/decrement operation.

3.4 Code Generation

With the table of assignments in hand, the final step was generating assembly code from the IR. The assignments provided the memory locations from which the variables could be loaded if they were not already in a data register. The AVIV compiler uses the architecture given by the ISDL description to generate assembly code. However, for the sake of simplicity the targeted architecture is fixed, with register indirect and auto-increment/decrement loads and stores available. Since memory layout optimization would occur as an optimization pass within the AVIV compiler, there is no loss of generality by assuming the target architecture at this point, since the AVIV compiler already creates mappings between SUIF operations and the operations available in the target ISDL architecture [3][16].

3.4.1 Overview of Target Architecture

The target architecture for assembly code generation assumed for this thesis has the following features:

- A separate instruction and data memory. Only the address registers in the AGU are allowed to address the data memory.
- An address generation unit (AGU). The AGU has a small register file for the address registers. Only address registers can be loaded with constants from the instruction stream.
- An arithmetic logic unit (ALU).
- A register file for the data registers. The data registers can store the result of the ALU, the data memory (for loads and stores), and the instruction memory (for loading constants).

The following figure illustrates the organization of the target architecture:

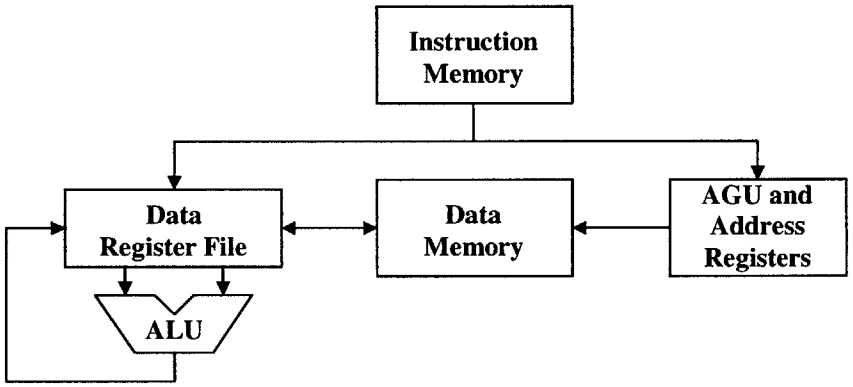

Figure 3.6: Target architecture organization.

The architecture is similar in nature to common, load-store RISC architectures.

3.4.2 Generating Assembly Code

As mentioned before, the IR supports a low-level representation that allows relatively straightforward translation into assembly code. The simple three-operand

instructions in the IR can often be directly mapped to a matching instruction in the target architecture. However, the operands of the instructions need to be loaded before they can be used. Since register allocation is part of code generation, a scheme for allocating registers is required. A simple solution of rotating the next available register in round-robin fashion among the data registers was implemented. Using a more sophisticated register allocation scheme is unnecessary since the purpose of generating assembly code is to evaluate the number of instructions generated to load address registers or perform address arithmetic. The results of operations are immediately stored to memory after calculation, so that the data in a register can be simply overwritten when the register is needed, whether or not it is already in use. The register to which a result is written is recorded in a symbol table, so that a subsequent operation which needs the result can avoid a load operation by first checking the symbol table to see if the value is already in a register. When a register is overwritten, the symbol table must be appropriately updated so the next time the overwritten data is needed, a load instruction will be generated.

Generating assembly code for branch structures is only slightly more difficult than for the three-operand instructions. As mentioned in section 3.2, each part of the branch structure maintains a separate list of instructions. These lists themselves are composed mainly of three-operand instructions, though nesting of if-statements within the header portion of the IF structure is allowed. Since generating code for the three-operand instructions is straightforward, the only difficulty is to generate them in the right order. For example, in an IF structure, code implementing the header would be generated first, followed by the then-part, and finally the else-part. Labels are generated to allow branching around the code not to be executed, such as code for the else-part if the header

evaluated to true. Nested IF structures within the header or any other instruction list is handled as any other branch structure and poses no difficulties.

The final detail in generating assembly code is the handling of address registers at basic block boundaries. Branch structures provide different paths for the control flow to follow. The generation of instructions for loads, stores, and address pointer manipulation depend on the values in the address registers, which may be different depending on which control flow path was followed. There are two cases that need to be considered: divergence of control flow from one path, and convergence of control flow from some number of different paths to one path. For each case, a conservative approach would be to assume nothing is known about the values in the address registers, and to generate code that loads each address register with the address of its next memory reference. However, this leads to an unacceptable number of address register loads.

Some observations can be made regarding control flow divergence or convergence that helps reduce the number of address register manipulation instructions. In the first case, each divergent path needs to be given the exact same values of the address registers, since each path starts from the same "state," which encompasses the values of all address registers, the next available register number, and symbol tables. All paths would then be able to use existing values of the address registers, as well as the values in the data registers. In the second case, each control flow path has its own "state," which needs to be consolidated. The equivalent address registers for each path can be compared, and if they all have the same value, then we can set the consolidated version of that address register to that value. If the compared values are not the same, then during code generation process the next use of the address register will force a load address register

instruction to be generated. The symbol tables are more difficult to compare, so the conservative approach of marking all symbols as not residing in registers is taken here. Since all registers have been effectively "cleared" the next available register can be safely set to any data register. The extra effort to duplicate and consolidate "states" for control flow paths is minimal, and gains extra assembly code compactness.

Chapter 4

Results

4.1 Code Generation Results

Assembly code was generated for several C code segments, ranging from small code segments with a limited number of variables to longer code segments with a greater number of variables and a number of control flow changes.

This implementation of optimizing memory layout lacks capability to handle while-loops and for-loops properly, due to a lack of time, though it handles if-statements. Adding the capability to handle these other types of control flow would be time-consuming but not difficult, as the basic mechanism for handling changes in control flow are in place.

Table 4.1 shows the results of running the example programs with varying numbers of address registers. The figures in columns headed with "non-addr" represent the number of instructions that are not address register modifying instructions, and the figures in columns headed with "addr" represent the number of instructions needed for modifying the address registers. The "ratio" is computed by (non-addr + addr)/addr, and is an indication of the overhead introduced modifying the address registers. The "*" in the "addr" column indicate that between n and n+1 address registers available, the

	non-addr	Number of Address Registers Available					
		1		2		3	
Example		addr	ratio	addr	ratio	addr	ratio
ex1	11	2	1.182	2	1.182	2*	1.182
ex2	13	1	1.077	1*	1.077	---	---
ex3	11	1	1.091	1*	1.091	---	---
ex4	20	3	1.15	3	1.15	3*	1.15
ex5	26	2	1.077	2*	1.077	---	---
ex6	38	5	1.132	5*	1.132	---	---
ex7	35	1	1.028	1*	1.028	---	---
ex8	25	1	1.04	1*	1.04	---	---
ex9	24	1	1.042	1*	1.042	---	---
ex10	23	1	1.043	1*	1.043	---	---
ex11	26	3	1.038	3*	1.038	---	---
ex12	34	7	1.206	5	1.147	5*	1.147
ex13	20	6	1.3	3	1.15	3*	1.15
ex14	28	9	1.321	6	1.214	6*	1.214
ex15	32	9	1.281	7	1.054	7*	1.054
ex16	44	14	1.318	9	1.204	9	1.204

Table 4.1: Results of optimizing memory layout using GOA.

	non-addr	Number of Address Registers Available					
		1		2		3	
Example		addr	ratio	addr	ratio	addr	ratio
ex1	11	2	1.182	2	1.182	2	1.182
ex2	13	3	1.231	2	1.154	1	1.077
ex3	11	3	1.273	2	1.182	1	1.091
ex4	20	6	1.3	3	1.15	3	1.15
ex5	26	4	1.154	2	1.077	3	1.115
ex6	38	5	1.132	2	1.053	3	1.079
ex7	35	3	1.086	2	1.057	3	1.086
ex8	25	3	1.12	2	1.08	3	1.12
ex9	24	1	1.042	2	1.083	2	1.083
ex10	23	1	1.043	2	1.087	2	1.087
ex11	26	5	1.192	2	1.077	3	1.115
ex12	34	6	1.111	9	1.265	4	1.118
ex13	20	6	1.3	5	1.25	4	1.2
ex14	28	7	1.25	9	1.321	6	1.214
ex15	32	9	1.281	6	1.188	7	1.219
ex16	44	12	1.273	10	1.227	10	1.227

Table 4.2: Results from generating code without memory layout optimization.

additional address register was not used, as the cost of using the additional address register outweighed the cost of not using it. There are some entries in the table that indicate that adding an additional address register had the same cost as using one fewer registers. Overall, most code examples did not show any improvement when using more than one address register. This could be due to a number of factors, including little reuse of variables in the code segments, or possibly ineffective partitioning. Of the examples that did show improvement, the amount improvement was in the range of 5 to 10 percent, with one example as high as approximately 11 percent.

4.2 Comparison with Code Generated without Optimization

For comparison, Table 4.2 contains the results from code generated without performing memory layout optimization. In the situation where there was more than one address register, the program variables were split fairly evenly between the available address registers. The layout assignment was based on program order.

The data shows erratic behavior in terms of the number of instructions generated for address register manipulation. In most cases, using only one address register leads to a greater proportion of the code being address loads or address arithmetic, which is expected. When two address registers are used, however, the data indicates mixed results. For some of the examples, there is some improvement in the ratio of total instructions to non-address register instructions, in the range between 5 and 10 percent as seen before. However, some of the examples had worse ratios, which may be explained by the partitioning method. The variables are split as evenly as possible between the two partitions, with each partition alternately getting the next new variable seen in program

order. The cases in which the ratios get worse likely indicate instances where this partitioning method does not work well with the actual access sequence of the example.

When three address registers were used, there is an indication that in some cases the number of instructions generated to perform address loads or address arithmetic is no longer dictated by the program's access sequence, but rather by the forced initialization of all three address registers because of the partitioning of the program variables. For example, Figure 4.1 compares generated assembly code without optimization for a code sequence, using two and three address registers. The SETAR() instructions are used to load the address of a variable into an address register. The assembly code for two address registers contains only two SETAR() instructions, one for loading each of the two address registers, while the assembly code for three address registers contains three SETAR() instructions, one for loading each of the three address registers. This occurs in code sequences for which the number of variables is small. For the larger examples where the number of address instructions remained greater than the number of address registers, there is generally improvement in the address instruction overhead.

Code sequence

```
int i;
int a,b,c;

i = 2;
a = 1;
b = 1;
c = 7;

c = c - i;
a = a+c;
b = a*b;
i--;
c = c - i;
a = a+c;
b = a*b;
i--;

return (b);
```

assembly code for 2 address registers

```
LDC( 2, R0 )
LDC( 1, R1 )
LDC( 1, R2 )
LDC( 7, R3 )
SUB( R3, R0, R3 )
SETAR( 2, AR1 )
ST_INC( R3, AR1 )
ADD( R1, R3, R1 )
ST_DEC( R1, AR1 )
MUL( R1, R2, R2 )
SETAR( 1, AR0 )
ST_DEC( R2, AR0 )
LDC( 1, R4 )
SUB( R0, R4, R0 )
ST_INC( R0, AR0 )
SUB( R3, R0, R3 )
ST_INC( R3, AR1 )
ADD( R1, R3, R1 )
ST( R1, AR1 )
MUL( R1, R2, R2 )
ST_DEC( R2, AR0 )
LDC( 1, R5 )
SUB( R0, R5, R0 )
ST( R0, AR0 )
RET( R2 )
```

assembly code for 3 address registers

```
LDC( 2, R0 )
LDC( 1, R1 )
LDC( 1, R2 )
LDC( 7, R3 )
SUB( R3, R0, R3 )
SETAR( 2, AR1 )
ST_INC( R3, AR1 )
ADD( R1, R3, R1 )
SETAR( 0, AR0 )
ST( R1, AR0 )
MUL( R1, R2, R2 )
ST_DEC( R2, AR1 )
LDC( 1, R4 )
SUB( R0, R4, R0 )
SETAR( 4, AR2 )
ST( R0, AR2 )
SUB( R3, R0, R3 )
ST_INC( R3, AR1 )
ADD( R1, R3, R1 )
ST( R1, AR0 )
MUL( R1, R2, R2 )
ST( R2, AR1 )
LDC( 1, R5 )
SUB( R0, R5, R0 )
ST( R0, AR2 )
RET( R2 )
```

Figure 4.1: Forced initialization of an unnecessary address register.
When there are two address registers available, only two address register loads are needed. An additional address register causes a different partitioning of the program variables, and the resulting assembly code is larger than before due to initialization of the additional address register.

Chapter 5

Conclusions

This thesis shows that the AVIV retargetable compiler can be designed so that it optimizes the memory layout of program variables, and that such optimization leads to assembly code that reflects an effective memory addressing strategy. The code generated by AVIV for architectures that have limited addressing modes, and support for auto-increment and auto-decrement instructions, will have reduced size if the optimization presented in this thesis is used, compared to the size of the code without the optimization. Though this optimization was not directly integrated into AVIV, it works on the same SUIF IR used by AVIV, and can be performed as an early step before the IR is used by the SPAM compiler [16]. The generated memory layout assignment can be used during the covering of the Split-Node DAGs to generate address register load and arithmetic instructions, which can then be scheduled by AVIV's code scheduling algorithms. For target architectures that have support for register+offset addressing or lack auto-increment and auto-decrement instructions, the optimization would be unnecessary as in the first case variables can be accessed directly using a frame-relative pointer and an offset, and in the second case address register load and arithmetic instructions would be needed no matter what the variable layout.

The efficacy of the optimization method, based on solving GOA, is clearly shown by the results from generating assembly code from segments of C code. With one address register in use, omitting the optimization leads to more instructions devoted solely to adjusting the address register for most examples. When more address registers are added, there is still a general trend showing that using the optimization leads to a smaller size for generated assembly code. With multiple address registers in use, the method of partitioning the variables among the address registers becomes a determining factor in the final size of generated assembly code. A good partitioning method will lead to smaller assembly code size consistently, while a more inferior method will exhibit erratic performance on different code samples and be much more dependent on the particular access patterns in the C code for its gains in reducing instruction count.

5.1 Future Work

There remain several areas in which the optimization developed here can be improved, and thus making the optimization more useful in AVIV:

1. Adding capability to handle while-loops and for-loops. As discussed in Chapter 4, adding the capability is not difficult with the means of handling control flow change in place, though the handling of while and for loops is slightly different from the three-operand instructions and if-statements.
2. Adding capability for increment/decrement sizes larger than 1. Many embedded processors and DSPs have address generation units capable of modifying addresses more than ± 1. There has already been research into this area, and several methods proposed in [11] and [14].

3. Improving the partitioning scheme. Since the proper partitioning so directly affects the number of address register instructions, implementing a more optimal partitioning scheme, such as suggested in [10], may improve things.
4. Improving the selection of edges during SOA. Currently there is no strategy in selecting among edges with the same weight in SOA when deciding to include or exclude an edge from the access graph. One strategy that might lead to consistently better disjoint path covers is during selection of an edge from a group of edges with equal weight, the edge which has the greatest number of vertices currently not covered in the cover graph should be selected. If after applying this criterion there remain multiple eligible edges, then any of the remaining edges may be selected.

Bibliography

[1] G. Hadjiyannis. *ISDL: Instruction Set Description Language -- Version 1.0*, MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge, Massachusetts. November 1998. (http://www.caa.lcs.mit.edu/~ghi/Postscript/isdl_manual.ps)

[2] G. Hadjiyannis, S. Hanono, and S. Devadas. "ISDL: An Instruction Set Description Language for Retargetability," *Proceedings of the 35th Design Automation Conference*, pp. 299-302, June 1997.

[3] S. Hanono and S. Devadas. "Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator," *ACM/IEEE Design Automation Conference*, 1998.

[4] Stanford Compiler Group. *The SUIF Library – Version 1.0*, 1994.

[5] D. H. Bartley. "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes," *Software — Practice and Experience*, vol. 22(2), pp. 101-110, February 1992.

[6] G. Araujo, A. Sudarsanam, and S. Malik. "Instruction Set Design and Optimizations for Address Computation in DSP Architectures," *Proceedings of International Symposium on System Synthesis*, 1996.

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, G. Araujo, A. Sudarsanam, S. Malik, V. Zivojnovic and H. Meyr. "Code Generation and Optimization Techniques for Embedded Digital Signal Processors," *Proceedings of the NATO Advanced Study Institute on Hardware/Software Co-Design*, 1995.

[8] S. Liao. "Code Generation and Optimization for Embedded Digital Signal Processors," Ph.D. Thesis, MIT Department of EECS, January 22, 1996.

[9] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. "Storage Assignment to Decrease Code Size," *ACM Transactions on Programming Languages and Systems*, vol. 18(3), pp. 235-253, May 1996.

[10] R. Leupers and P. Marwedel. "Algorithms for Address Assignment in DSP Code

Generation," *ACM/IEEE International Conference on Computer-Aided Design*, pp. 109-112, 1996.

[11] A. Sudarsanam, S. Liao, and S. Devadas. "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures," *Proceedings of ACM/IEEE Design Automation Conference*, 1997.

[12] N. Sugino, H. Miyazaki, S. Iimuro, and A. Nishihara. "Improved code optimization method utilizing memory addressing operation and its application to DSP compiler," *International Symposium on Circuits and Systems*, pp. 249-252, 1996.

[13] N. Sugino and A. Nishihara. "Memory allocation methods for a DSP with indirect addressing modes and their application to compilers," *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 2585-2588, 1997.

[14] N. Kogure, N. Sugino, and A. Nishihara. "DSP memory allocation method for indirect addressing with wide range update operation by multiple registers," *IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 435-438, 1998.

[15] N. Sugino, H. Funaki, and A. Nishihara. "Memory address allocation method for a indirect addressing DSP with consideration of modification in local computational order," *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 496-499, 1999.

[16] Hanono, S. "Aviv: A Retargetable Code Generator for Embedded Processors," Ph.D. Thesis, MIT Department of EECS, June 1999.

[17] Liem, Clifford. *Retargetable Compilers for Embedded Core Processors.* Kluwer Academic Publishers, 1997.