FREELIST-BASED STACK FRAME ALLOCATION

BY

THOMAS RICHARD NOVAK

B.S., University of Illinois at Urbana-Champaign, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

ABSTRACT

This thesis contributes an updated method of stack frame allocation that eliminates false dependences between sibling procedure calls by guaranteeing that the stack space occupied by each is mutually exclusive. To make this guarantee, frames are allocated as in a queue or freelist, and the corresponding memory is managed by an efficient stack compactor. Though this method increases dynamic instruction counts by an average of 7.5%, we show that this is necessary to enable a sufficiently inclined execution paradigm to effectively disseminate parallelism. We also show that the memory management required has little to no impact on performance. Furthermore, freelist-based stack frame allocation is shown to succeed in reducing stack stores with antidependences of distance less than 100 dynamic instructions from 27% of all program stores to 3.5%, effectively renaming stack frames.

To Mom and Dad.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my parents, Richard and Barbara; without their hard work, sacrifice, and endless support, I would not have had the strength and wherewithal to accomplish what I have.

Thanks to John Murphy and Anil Pandya at Advanced Micro Devices for being understanding about my drop in productivity during this undertaking. Thanks to Mike Fertig also at AMD for providing me with an honest review, and for his assistance with the technicalities of thesis presentation. Thanks to Marika Constantaras at UIUC for being my on-campus agent while I was most decidedly off-campus, and for her endless moral support.

A special thank-you to my advisor Matt Frank at UIUC for his wisdom, patience, understanding, and encouragement. Even though the latter part of our collaboration has taken place at a considerable distance, I am thoroughly convinced that my graduate school experience would not even have been half as inspiring or fulfilling with anyone else, under any other circumstances.

TABLE OF CONTENTS

Page

CHAPTER 1

INTRODUCTION

For decades, computer architecture has been driven by the producer-consumer computational model of instruction supply and execution. The effectiveness of a particular machine is inexorably linked to this relationship; the enhancement of one cannot improve performance without adequate attention to the other. The body of research on either end is vast but also in constant metamorphosis, requiring perpetual re-evaluation of enduring paradigms. In the case of instruction fetch, research trends are relentless in pursuit of increasingly aggressive mechanisms to identify and exploit parallelism, such as the disbursal of multiple threads of execution upon the machine. This thesis relates to the effect such fetch mechanisms have on execution.

Parallel execution relies heavily on the recognition and elimination of false dependences between operands of all kinds. In modern superscalar processors, a large number of provisions exist to eliminate and reduce dependences that result from the serial-nature of computer programs [1]. Register renaming, for example, eliminates output and antidependences with respect to register usage. Also, to reduce the penalties associated with true dependences in registers, bypass paths are added to the hardware that enable the forwarding of pertinent results to subsequently executing instructions. In addition to these performance enhancements, further progress can be made reducing dependences in memory accesses. Some constructs utilized today include associative store queues with store/load forwarding. A store queue essentially implements bypass paths for memory operands, as well as provides renaming to break false dependences between them. In future processors with high instruction level parallelism (ILP), ever increasing instruction window sizes introduce more simultaneously in-flight memory instructions with potentially matching addresses, and therefore create a need for scalability in bypassing mechanisms [2]. Efforts to scale such mechanisms would

theoretically benefit from a reduction in the false dependences they are designed to accommodate. In an examination of these dependences, this thesis will show that a significant amount are associated with the way in which procedure activation frames are currently allocated on the stack, and therefore could be significantly reduced by a modified scheme.

According to the current methodology, when a procedure is called, its frame will occupy many if not all of the same addresses on the stack as the frame of the previous call, including the load and store addresses used to maintain callee-saved registers. Therefore, in a particular call, the target addresses of the stores that save those registers will alias with the addresses of the loads that restored them in the previous call. If the distance between these aliasing instructions is small, the false dependences will make the stores wait to issue until the preceding loads have issued. Or, in a more aggressive design, the stores could be allowed to execute out-of-order, which would force any subsequent load to perform a timestamp lookup and associative table read to find the appropriate store from which to obtain its result. Either way, if it were guaranteed that accesses to a particular stack frame could never overlap with those of another, false dependences like those described above would be broken, effectively renaming that frame. The main contribution of this thesis is the introduction of a compilation scheme which makes that guarantee.

CHAPTER 2

MOTIVATION

As instruction fetch mechanisms become more aggressive and allow consideration of an increasingly large set of instructions from more locations in a program for execution, interprocedural dependences will likely play a larger role in restricting available parallelism. This chapter motivates this claim with respect to the pervasiveness of a particular kind of interprocedural dependence, as well as outlines a class of fetch mechanisms in which the detrimental effect of this dependence is paramount.

2.1 Stack Antidependences

To reduce false dependences between memory operands, it may be desirable to rename stack frames such that they occupy unique addresses with respect to one another. As a motivation for this type of scheme, it is necessary to gain insight into how often aliasing occurs between stack frames. If the occurrence of aliasing is substantial, then eliminating the dependence could be beneficial.

The most likely dependence that occurs due to the current frame allocation paradigm is between the last few instructions of a returning procedure, and the first few of the next procedure. The last few instructions of any procedure call pop callee-saved registers off the stack, whereas the first few instructions are when those registers are pushed. If the number of instructions between these two events is small, it is likely to cause antidependences between the current stack frame's store instructions to push registers and the old frame's load instructions to pop them. The seven benchmarks outlined in Chapter 4 were run to determine the frequency and extent of this behavior.

Overall, 51.5% of stores were found to have destination addresses on the stack. Of those

stores, 52.6% had an antidependence with a load that occurred less than 100 instructions prior. The average distance of that antidependence, which occurs in approximately 25% of all stores, was found to be 31.9 instructions. Figure 2.1 is a histogram of dependence distances averaged over all the benchmarks. The bars labeled "Observed" indicate the true histogram, whereas the bars labeled "Cumulative" indicate the total percentage running from left to right. For example, 85% of antidependence distances less than 100 instructions on the stack have a value less than 50 instructions. Approximately 50% of antidependence distances less than 100 instructions on the stack have a value less than 30 instructions. In summary, approximately one-eighth of all program stores are to the stack and have an antidependence with a preceding load with distance less than 30 instructions. This suggests that interprocedural stack frame aliasing is indeed significant, and could therefore benefit from a modified allocation scheme that eliminates those dependences.

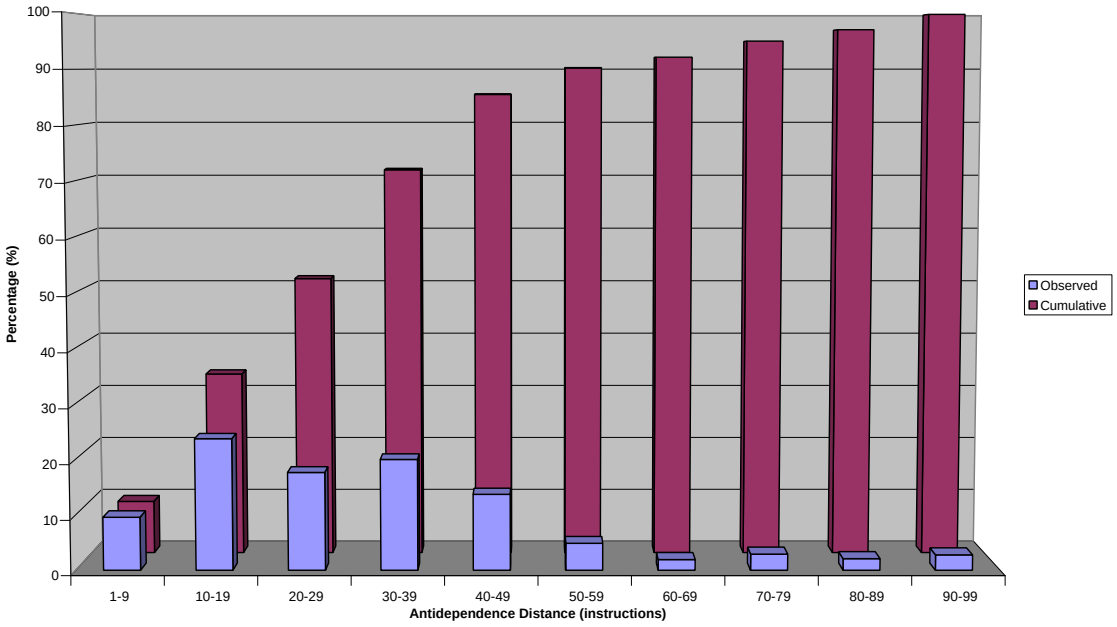


Figure 2.1: Histogram of stack store antidependence distances less than 100 instructions averaged over the seven SPECint benchmarks outlined in Chapter 4. Stack stores with antidependences of distance less than 100 instructions represent 27% of all program stores. Approximately one-eighth of all program stores are to the stack with an antidependence of distance less than 30 instructions.

2.2 Multithreading

Increasingly aggressive fetch mechanisms illuminate the previously benign construct of stack frame allocation, which unnecessarily maintains data dependences between procedures and therefore inhibits available parallelism. Multithreading is an example of such an aggressor, where instruction supply is increased by fetching from multiple threads of execution. Simultaneous multithreading [3] allows for execution of instructions from multiple independent threads or programs. It is dependent threads from a single program, though, that provide motivation for stack frame renaming. The way in which these threads are spawned and contend with register and memory dependences defines each particular implementation.

In one proposed implementation called dynamic multiple threads (DMT) described in [4], this speculation occurs at procedure call points and backwards branches, where a thread is spawned to continue executing the current procedure while the main thread handles the call or branch. This method utilizes data prediction to alleviate false dependences created when callee-saved registers are pushed to the stack, but does not address dependences between leaf or sibling procedures (same caller). DMT also does not employ any compiler assistance in achieving its throughput.

Another example of executing multiple dependent threads is the multiscalar architecture [5, 6]. This architecture executes "tasks" identified by a static walk of the control dependence graph, which are then distributed to processing units organized as a queue. These units handle memory dependences by speculation, utilizing an address resolution buffer (ARB) [7], and by passing data forward in the queue as needed. The tasks assigned by the sequencer may include any number of constructs such as loop iterations, multiple basic blocks, or procedures. In the case of parallel execution of procedure calls, interprocedural memory dependences are left to the ARB.

These examples show that procedures are often chosen as threads, and significantly reducing interprocedural dependences caused by the traditional stack frame allocation paradigm may be desirable. Evaluating a new frame allocation technique in a multithreaded environment is beyond the scope of this thesis; however, some insight into the potential benefits of such a technique with respect to increased parallelism in a single-threaded superscalar machine is provided in Chapter 5.

CHAPTER 3

DESCRIPTION

In order to implement a new stack frame allocation scheme, a suitable production compiler had to be chosen. As described in Chapter 4, GCC 2.8.1 configured for 32-bit MIPS was already available in our tool set, so it became the host of the changes in design described throughout the following sections.

3.1 Stack Frame Allocation

A stack frame or procedure activation frame is a collection of locations on the program stack devoted to the temporary storage required by a particular procedure call. Stack frames are used for many reasons, including to pass arguments from caller to callee, as temporary storage for a procedure that needs to do register spilling, as storage for stack-allocated data structures, and as a place to push temporary registers in order to restore them upon returning control to the caller. These frames are created dynamically at the beginning of a procedure call in a section of code called the prologue. The prologue typically allocates space for the frame by subtracting the statically determined frame size from the stack pointer register. Then, callee-saved registers are stored into the new frame locations, and the frame pointer is updated to match the new stack pointer. Since it is common for organizational and functional purposes for procedures to call other procedures, any number of stack frames can be active on the stack at any given time. The most active frames a program will have is equal to the depth of its dynamic call tree. In addition to what a frame contains, when it is allocated, and how many there are, there are two factors that complete a particular allocation scheme: (i) how the frames are organized relationally, and (ii) how the stack space is managed.

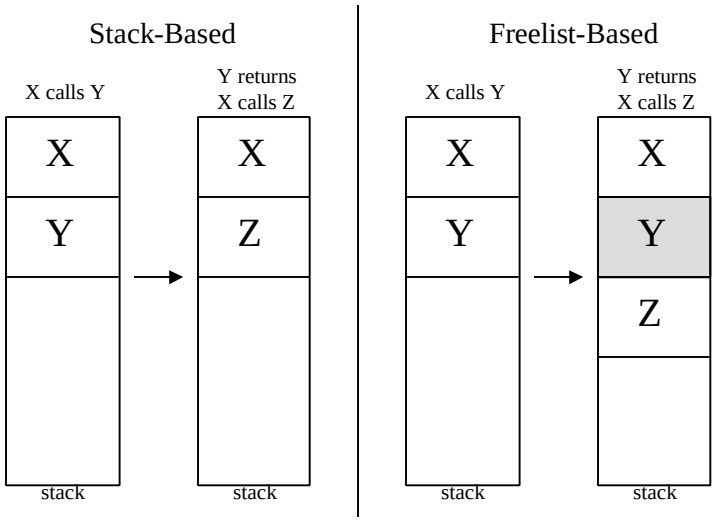


Figure 3.1: Frame allocation schemes.

Two different methods that address those issues are shown in Figure 3.1 and described below.

3.1.1 Stack-based

Stack-based allocation is so named because of the scheme's memory management. Upon completion of the procedure's task, a portion of code known as the epilogue is called. The epilogue loads the registers that were stored in the prologue, restores the stack pointer to its original value, and returns control to the caller. This effectively deallocates (or pops) the callee's frame, and when another procedure is called, its frame is allocated (or pushed) into the same space, as shown on the left side of Figure 3.1. From an organizational standpoint, this means that a procedure's frame is always adjacent to its caller's frame, and therefore the active call chain is represented by one contiguous block of frames on the stack.

3.1.2 Freelist-based

In freelist-based allocation, which is the contribution of this thesis, the epilogue loads temporary registers as above, but does not restore the stack pointer. In fact, this scheme calls for a departure from the traditional view of the stack pointer, towards the concept of a freelist pointer. After a procedure returns control to its caller, the corresponding frame has been left on the stack as

garbage, and the stack (freelist) pointer now points to the next available location for a subsequent frame. When the next procedure is called, its frame will be placed in that location, as in the right side of Figure 3.1. Unlike the stack-based method, frames are now self-contained in that they are not required to be adjacent to their caller's frame. What this method lacks in comparison to stack-based, however, is built-in memory management. With old frames left on the stack as garbage, addresses will be used up quickly and the stack may overflow. In order to prevent this, freelist-based allocation includes a stack compaction mechanism that is described below.

3.2 Stack Compaction

This section describes a memory management technique for freelist-based frame allocation called stack compaction, which arises out of the lack of deallocation of frames as part of the normal execution of that scheme.

3.2.1 Necessity

With dead frames left on the stack as garbage in freelist-based allocation, procedure-intensive or highly recursive programs will experience a significant increase in stack size. As system memory becomes more and more plentiful, this increase in size will most likely not negatively affect the average case. However, in order to prevent the stack from overflowing its boundary when such a boundary is to be reckoned with, stack compaction is introduced. To enable compaction, the system must be able to detect an inherent boundary overflow in the process of allocating a stack frame, and invoke the compaction algorithm accordingly. This can be accomplished by either trapping into the operating system, or adding a conditional branch to the prologue. Upon returning from the compaction algorithm, the initiating frame is allocated (unless the particular implementation allocated it before compaction) and execution resumes. Since the preceding description involves filling the pipeline with non-program-related instructions, it is natural to assume that performance may suffer as a result. A treatment of this issue is to follow in Chapter 5.

3.2.2 Algorithm

A conceptual view of the stack compaction algorithm is given in Figure 3.2. The algorithm consists

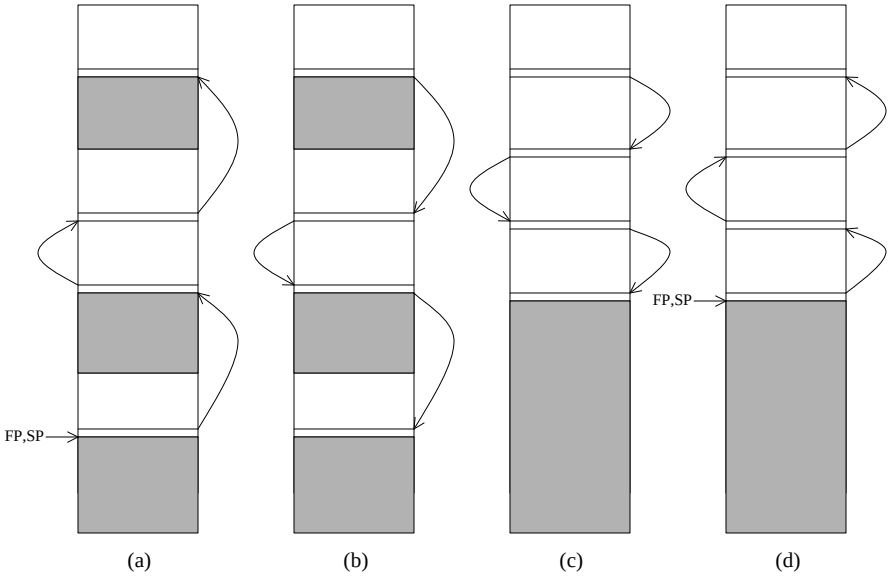


Figure 3.2: Stack compaction steps from (a) original stack, to (b) pointer reversal, to (c) copying, to (d) pointer restoration.

of three parts: (i) pointer reversal, (ii) copying, and (iii) pointer restoration. This description will assume that the system stack limit is such that it allows for the allocation of the frame that tripped compaction before compaction takes place. This means that the stack pointer and frame pointer are equal at this time, as shown in Figure 3.2(a). Furthermore, it is assumed that the frame size may not have been stored to the new frame before compaction is called. Subsequently, the compactor takes a pointer to the newly allocated frame and its frame size as input.

Figure 3.2(a) shows an example stack before compaction is called. From the figure, it is clear that freelist-based allocation leaves the active frames on the stack looking exactly like, at any given time, a singly linked list through the frame pointer chain. Compaction is therefore extremely easy and efficient, with run-time on the order of the number of active frames, and needing no additional storage to complete. This is in contrast to traditional copying garbage collection, which requires additional storage equal to the size of the area being collected. One list traversal is performed to reverse the pointers between frames as shown in Figure 3.2(b). With this configuration, copying consists simply of moving each frame at the other end of the current pointer word-by-word (if that frame is not already adjacent to the current one, which it may be), and then updating the pointer. Then, another pointer reversal is performed, and the algorithm terminates. Figure 3.2(c,d) shows

these last two steps. The return value of the algorithm is the new pointer representing the frame and stack pointers as in Figure 3.2(d). The actual C code for the stack compaction algorithm appears as an Appendix. From that code it is clear that the compactor relies on what it knows about stack frame layout, as described in the next section, in order to find pointers and frame sizes.

3.3 Frame Layout

In order to accomplish freelist-based stack frame allocation, several changes to the layout of each stack frame are required. In many cases, these changes introduce either additional storage or code overhead which is outlined in Section 3.4. The extent of this overhead will be evaluated in Chapter 5. Figure 3.3 illustrates the layout differences between allocation schemes, while their descriptions follow.

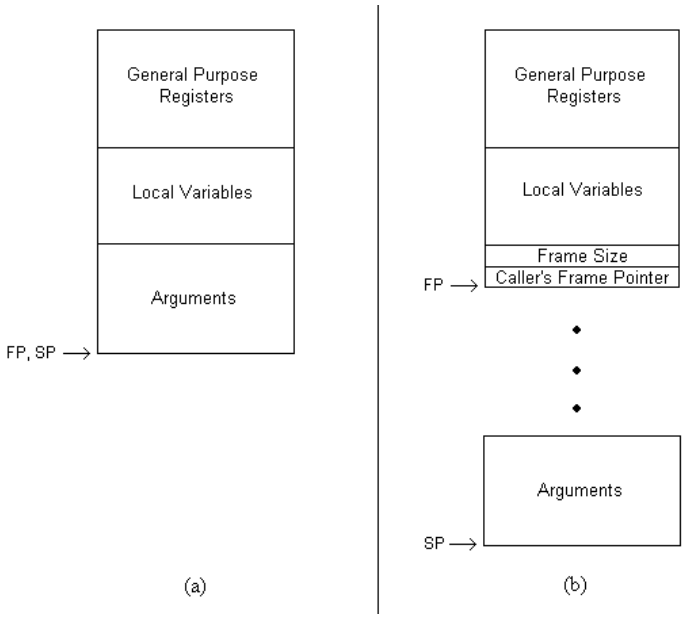


Figure 3.3: Stack frames from (a) stack-based and (b) freelist-based allocating compilers.

3.3.1 Stack pointer versus frame pointer

Most RISC architectures have registers devoted to the stack and frame pointers. In the traditional stack-based frame allocation scheme, the frame pointer is rarely needed; since any call made by a procedure will allocate then deallocate its frame, the procedure itself can assume that at all relevant

times, the stack pointer points to its own frame. Exceptions to this include procedures with stack-allocated data structures, such as in the case of the C function `alloca`. The compiler recognizes the different cases, and in procedures where the frame pointer is unnecessary, all references to the stack frame with the frame pointer as a base register are modified to use the stack pointer register instead.

In freelist-based allocation, however, upon return from the first call a procedure makes, the stack pointer will no longer point to the caller's frame. Instead, it will point to the next available location for a stack frame to be placed. Therefore, references to any procedure's stack frame must use the frame pointer as a base register. Furthermore, each frame must contain the value of its caller's frame pointer, so it can be restored upon return. This frame pointer storage is indicated in Figure 3.3(b). It should be noted that the stack-based allocation frame of Figure 3.3(a) may also include the value of the caller's frame pointer register in the block labeled "General Purpose Registers" in the cases where a frame pointer is necessary as mentioned previously. In addition to that storage being required for every procedure with a stack frame in freelist-based allocation, it must also be moved to a constant offset from the frame pointer so that it is accessible to a stack compactor that has no knowledge of stack frame contents.

3.3.2 Calling convention

In stack-based frame allocation, a procedure's frame is guaranteed to be adjacent to its caller's frame on the stack. Taking advantage of this fact, argument passing between the two is relatively simple. When generating the code to load stack-passed arguments, the compiler calculates relative addresses by adding offsets to the current procedure's frame size, effectively indexing into the caller's frame. This is why in Figure 3.3(a) the block labeled "Arguments" is at the bottom of the frame. This block represents the largest argument block a procedure will need to pass to any other procedure. Before each call, stack-passed arguments are simply stored to this block, and the callee knows to find them there.

A direct implication of the lack of spatial relationships between frames allocated on a freelist demands a change in the calling convention used to convey these arguments. Stack-passing arguments as above would simply not work, since it is unknown what will immediately precede a procedure on the stack. There could be any statically indeterminable number of dead frames between caller and

callee. In other words, a compiler could not generate relative addresses to find incoming arguments because the distance between the two frames is a dynamically determined value. But, there is one value guaranteed to point to the newly allocated frame and that is the freelist (stack) pointer. Therefore, if the stack pointer is decremented and arguments are stored as offsets from that pointer right before the call, they will again be adjacent to the callee's stack frame and can be indexed as in the stack-based allocation case. This argument block migration affects the frame layout as shown in Figure 3.3(b).

Another implication of the positioning of stack frames has to do with the high-level programming language construct of pass-by-reference. When arguments are passed to a procedure by reference, a pointer to the caller's frame may be pushed into the callee's frame. In stack-based frame allocation this is allowed because frames never move; the points-to address will always be valid as long as the pointer itself is valid. In freelist-based allocation, however, memory management via stack compaction introduces the possibility that a pointer passed by reference becomes invalid when the frame it points to is relocated. Since the stack compactor knows nothing of the contents of the frames it is relocating other than the overall frame pointer chain and frame sizes, it cannot detect and repair a pointer passed by reference.

The reference problem also appears in the use of the C function `alloca`. This function dynamically allocates space on the stack of variable size by decrementing the stack pointer and keeping a pointer to the allocated block in a temporary register. In MIPS frames allocated by a stack-based scheme, parameter passing is accomplished despite an `alloca` allocation by copying the outgoing argument block to the new location of the stack pointer. Any procedure calls that come before or after the `alloca` block do not change this layout, since their frames and any other allocations are simply popped off the stack upon return. However, in the freelist-based case, procedure calls before and after such an allocation would leave the corresponding block in the middle of any number of garbage frames. This is fine during normal execution since a temporary register is devoted to maintaining a pointer to that block and parameter passing is accomplished through floating argument allocation as described above. However, if stack compaction were to occur while this procedure's frame is live, the compactor would have no way of knowing about the `alloca` block and could potentially copy over it. Therefore, to enable this allocation scheme, keeping references to other stack frames or to variable size stack allocations must be forbidden.

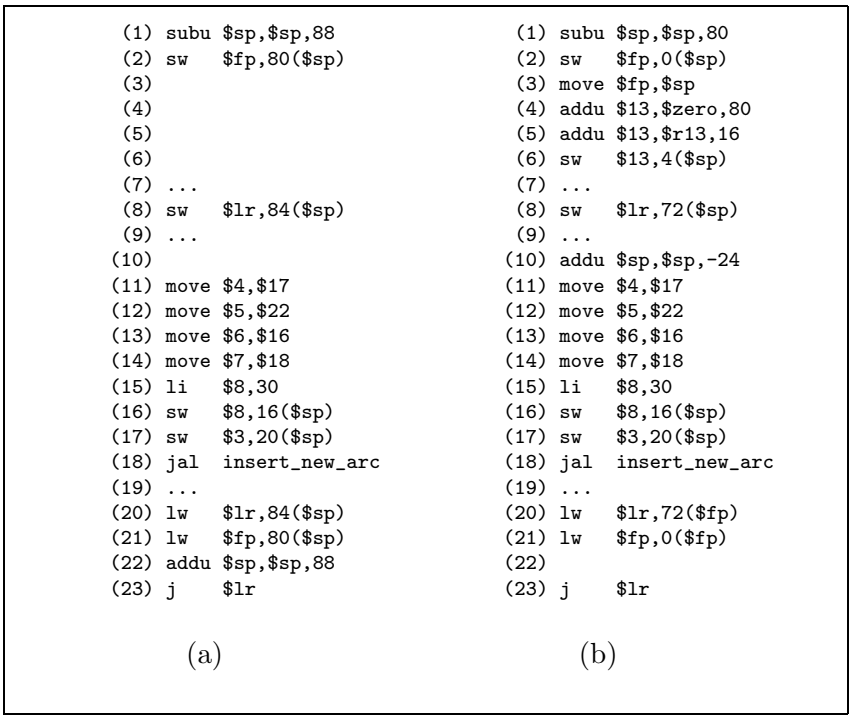
3.3.3 Frame size

During the pointer reversal and copy procedure described as stack compaction in Section 3.2, one piece of information is needed in addition to the pointers. Each frame must also know how many addresses it occupies on the stack to enable copying. This value is used during copying as the induction variable for the loop that copies a single stack location. The frame size contains two parts which are similar to Figure 3.3(b): (i) the size of its argument block, and (ii) the size of its main block including callee-saved registers, local variables, the frame size, and the caller's frame pointer. In the case of the argument block, that figure shows a frame and one of its outgoing argument blocks. When considering a stack frame for the purposes of stack compaction and therefore determining its size, it is the incoming argument block that is relevant. This block will always be adjacent to the rest of the frame per the calling convention described above, and it is therefore treated as a unit. The sum of these two sizes is stored as the overall frame size, at a constant offset from the frame pointer so the stack compactor can locate it.

3.4 Code Generation

All of the conceptual differences between the two frame allocation schemes manifest themselves in the compiler generated code in various ways. Figure 3.4 contains code segments from each compiler, for comparison. Recall that this study was done using GCC 2.8.1 for 32-bit MIPS. The code shown is an excerpt of the function `price_out_impl` from SPECint2000 benchmark 181.mcf. This particular function was chosen simply because it contains all the aspects of the two allocation schemes that this text has detailed thus far. To facilitate the comparison, most of the code has been removed and what appears has been reordered, while still maintaining correctness.

The very first line of Figure 3.4 shows different frame sizes. Certainly the freelist-based frame is smaller in part because it does not include the outgoing argument block like the stack-based frame does. In this case, that block size is 24 bytes, which does not match the difference between the two sizes on line 1 of the figure. There are more factors at play here, most of which have to do with the necessity of each storage section in MIPS stack frames to exist on 8-byte boundaries. For example, the frame pointer in the stack-based frame is stored into the general purpose register area (see Figure 3.3 for frame layouts), whereas in the freelist-based frame, it is not. However, the



```
(1) subu $sp,$sp,88
(2) sw   $fp,80($sp)
(3)
(4)
(5)
(6)
(7) ...
(8) sw   $lr,84($sp)
(9) ...
(10)
(11) move $4,$17
(12) move $5,$22
(13) move $6,$16
(14) move $7,$18
(15) li   $8,30
(16) sw   $8,16($sp)
(17) sw   $3,20($sp)
(18) jal  insert_new_arc
(19) ...
(20) lw   $lr,84($sp)
(21) lw   $fp,80($sp)
(22) addu $sp,$sp,88
(23) j    $lr
```

(a)

```
(1) subu $sp,$sp,80
(2) sw   $fp,0($sp)
(3) move $fp,$sp
(4) addu $13,$zero,80
(5) addu $13,$r13,16
(6) sw   $13,4($sp)
(7) ...
(8) sw   $lr,72($sp)
(9) ...
(10) addu $sp,$sp,-24
(11) move $4,$17
(12) move $5,$22
(13) move $6,$16
(14) move $7,$18
(15) li   $8,30
(16) sw   $8,16($sp)
(17) sw   $3,20($sp)
(18) jal  insert_new_arc
(19) ...
(20) lw   $lr,72($fp)
(21) lw   $fp,0($fp)
(22)
(23) j    $lr
```

(b)

Figure 3.4: Code from (a) stack-based, and (b) freelist-based stack frame allocating compilers.

size of the register area does not decrease for the freelist-based frame because removing 4 bytes causes the register storage to be unaligned, and therefore rounded up. Another related, interesting note is that the frame pointer in the stack-based frame is stored at all; it is not needed for this function. When GCC optimizes away the frame pointer and replaces it with the stack pointer, the frame pointer register is free to be considered for temporary register usage. This is why it is saved in Figure 3.4(a). Since the frame pointer is actually used as a frame pointer in the freelist-based frame, it has one fewer temporary register than the stack-based case, so it must allocate an additional 4 bytes of space in the local variable section of the frame. Since, in the particular case of the procedure in this figure, the stack-based frame has a local variable storage size of 24 bytes (aligned), adding another 4 bytes in the freelist case causes that storage to be unaligned, and therefore rounded up. Now all totaled, the freelist-based frame has 24 fewer bytes for outgoing arguments, no fewer bytes for registers because of rounding, 8 more bytes of local variables, and 4 more bytes for the frame size stored on line 6 in Figure 3.4(b) and shown in Figure 3.3(b) (rounded up to 8 bytes to maintain alignment), which equals 8 fewer bytes.

The necessity of the frame pointer also adds the copy instruction on line 3 in Figure 3.4(b).

The next three instructions are required to store the frame size in the frame itself, as required by stack compaction. As mentioned earlier in this chapter, there are two different parts to that overall size; line 4 contains the size of the frame also seen on line 1, and line 5 has the size of the incoming argument block for this function. Those values are added together and stored into the frame. It is interesting to note at this point that two instructions are needed just to add two numbers together in the case of frame size, because both those numbers are introduced into the machine as immediate values. This effect was not a point of concentration for this thesis.

Line 10 of the code snippets illustrates the other half of the change in calling convention. The first half was encountered in the discussion of the main frame size on line 1 in the form of the missing outgoing argument block in the freelist-based frame. Line 10 illustrates how the floating argument block is allocated just prior to its use for each call a procedure makes. In this case, the procedure `insert_new_arc` is being called, and requires 24 bytes (six arguments) of argument space. Lines 16 and 17 show two of those arguments being saved into the newly allocated block. The other 16 bytes of space are due to MIPS calling conventions; they are always allocated for any argument block (even if the space is never used) because MIPS requires that the maximum of four arguments it passes in registers have associated stack space. Those four argument registers are registers 4 through 7, as seen on lines 11 through 14. Therefore, lines 11 through 17 represent the six arguments being passed to `insert_new_arc`.

The only remaining disparity in the two code snippets relates to the differences in the allocation schemes. On line 22 of Figure 3.4(a), the frame is deallocated and the stack pointer is restored to its previous value. This instruction is absent from the right side of the figure, indicating that the frame will be left on the stack as garbage. Another interesting code-related note about freelist-based allocation is that the ordering of callee-save register restoration in the epilogue is important. The loads in the freelist-based epilogue need the frame pointer since the stack pointer is no longer relevant, yet the old value of the frame pointer itself must be restored. This frame pointer restoration must be the last load instruction before the procedure returns. This is not necessarily true in the stack-based case.

CHAPTER 4

METHODOLOGY

Before performing the various measurements that are desired to evaluate freelist-based frame allocation, an appropriate simulation environment is necessary. This chapter describes the parts of that environment, as well as some limitations with respect to its usage. Versatility of function and practicality of design time negatively interfere with respect to building a useful methodological environment. Some desirable yet currently unequipped features appear in Section 4.4.

4.1 Compiler

Since this thesis describes a new compilation technique, a suitable production compiler had to be chosen. However, this choice was not crucial; any versatile, configurable, reasonably well documented compiler would do. Our tool set already included GCC version 2.8.1 configured for MIPS, and after some investigation into MIPS calling conventions and stack frames, this compiler was chosen. Development within GCC required a somewhat steep learning curve, but once overcome, making changes to the stack frame allocation scheme was not exceedingly difficult. Most of the necessary changes were made to the ISA-specific configuration files. These files define many things about the ISA and code generation, including stack frame layout and calling conventions which are applicable here.

4.2 Simulator

At the heart of the simulation environment is a trace-driven simulator of the 32-bit MIPS ISA, which accepts and decodes MIPS executables. For the simulations which include cycle counts, a

built-in timing model was used. This model includes microarchitectural constructs appropriate to the studies being conducted. These constructs and their specifications are shown in Table 4.1.

Table 4.1: Processor parameters.

Parameter	**Value**
Fetch bandwidth	8 instrs. / cycle (peak)
Branch predictor	Tournament-style with 128K-entry pattern history table
Instruction window	512-entry
Issue width	8 instrs. / cycle
Register alias table	Alpha 21264-style
Store Queue	Fully associative, renaming
Reorder buffer	512-entry

4.3 Benchmarks

The program workloads used for evaluation and comparison consist of seven SPECint benchmarks. Their names and input sets are shown in Table 4.2. These benchmarks were not chosen to display or avoid anything specific; they were chosen because they were available and easy to port to MIPS. Therefore, their usage is only designed to provide insight into the particular behaviors they are used to generate, and not to be an authoritative or final treatment of those behaviors.

Table 4.2: Benchmarks and their inputs.

Benchmark	**Input Set**
129.compress	ref.in
164.gzip	lgred.program 1
176.gcc	mdred.rtlanal.i
181.mcf	test.in
197.parser	mdred.in
253.perlbmk	mdred.makerand.pl
256.bzip2	lgred.source 128

4.4 Limitations

In a perfect world, this environment would contain a few more features which it currently does not. First, freelist-based stack frame allocation by nature spreads out the address usage on the stack. This causes more replacement activity in the L1 D-cache which could evict useful information, for example heap data. The current simulator does not have the proper cache design to handle this case. Secondly, multithreading is discussed as both a motivation and target application of this freelist-based allocation, yet no simulation infrastructure exists in the current environment to substantiate any of those claims. Finally, the memory management aspect of freelist-based allocation requires the elimination of passing arguments by reference or dynamically allocating variable size data structures on the stack. This is also not currently supported. To supply a better context for consideration, these shortcomings will be mentioned again where appropriate throughout this thesis.

CHAPTER 5

EVALUATION

In order to gain insight into the pitfalls and possibilities introduced by freelist-based stack frame allocation, some quantitative analyses are desirable. The studies appearing below are chosen to be illustrative; they do not necessarily represent all possible costs, benefits, or applications relevant to this type of allocation.

5.1 Overhead

As mentioned in Chapter 3, there are two major costs to this new allocation scheme, which are described below. In order to leverage these against any potential benefits, it is necessary to gain a more detailed notion of their extent.

5.1.1 Dynamic instruction count

One of the main drawbacks to freelist-based frame allocation is the corresponding increase in dynamic instruction count due to the code generation requirements shown in Figure 3.4. The extent of this overhead can be determined from an instruction count generated by each of the seven benchmarks for each compiler. Figure 5.1 shows that count for each benchmark relative to the stack-based allocating compiler. As indicated in the figure, the increase in dynamic instructions executed ranges from 1% to 14%, with an average of 7.5%. Notice that 253.perlbmk has the second largest disparity in instructions at 13%; later it will be shown to respond the most to the new allocation scheme, with a noticeable increase in performance over the old scheme. These numbers are somewhat significant, but mostly necessary to achieve the stack frame renaming which is the

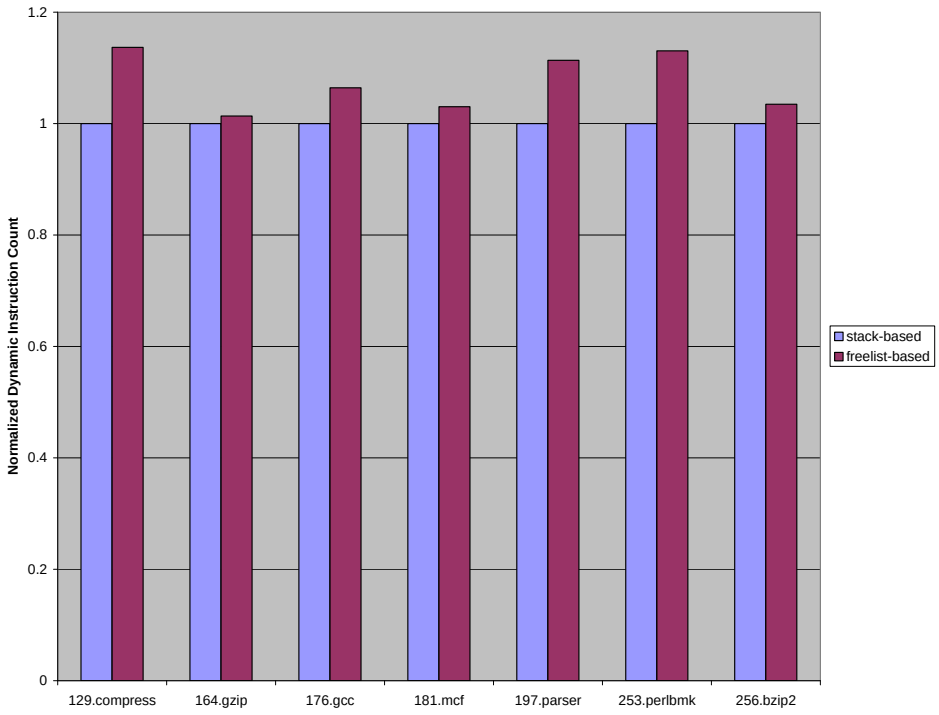


Figure 5.1: Dynamic instruction count in freelist-based allocation relative to stack-based. The average increase in this count over the seven SPECint benchmarks is 7.5%.

overall goal.

One unwanted side effect of this renaming was alluded to in Section 3.4 when GCC's ability to utilize an otherwise unnecessary frame pointer as a temporary register was discussed. Since the frame pointer is required for any procedure with a stack frame in freelist-based allocation, register spilling will be introduced in places it previously was not for stack-based allocation, contributing to the overall increase in instruction counts. This, as mentioned before, is not a direct product of the allocation scheme but rather due to a lack of registers that freelist-based allocation simply aggravates in some cases. Keeping an extra dedicated frame pointer register would eliminate this effect and drive the illustrated disparities in instruction counts down.

5.1.2 Stack compaction

Another potential cost to freelist-based allocation is the compaction required to manage the stack. With system memory becoming increasingly inexpensive, it was hypothesized earlier that this penalty would not be significant. This hypothesis is not possible to test directly, due to a lack

of support in the simulation environment for the elimination of pass-by-reference and variable size stack allocations which is necessary as described in Section 3.3.2. Without this elimination support, active stack frames cannot be relocated as they might contain data that is pointed to by other frames or the relocation might overwrite stack-allocated data structures. Is it also true that eliminating these stack references may come with a cost of its own, but since it is not currently supported, no evaluation of this cost will be presented.

In order to evaluate compaction as accurately as possible without actually utilizing it in real benchmarks, a conservative view of active stack frames was developed and applied to a simulated stack. The machine was assumed to impose a 128-MB stack overflow boundary, meaning that when a frame is allocated very close to this boundary, stack compaction is invoked. Furthermore, at the time of compaction, the stack was assumed to contain 16 4-KB stack frames, for a total of 64-KB of active stack space. Utilizing the current methodology with the active frame layout just described, the stack compactor was observed to require approximately 488 000 cycles to complete. Then, each benchmark was run to determine the number of times this compaction would be required.

Table 5.1: Stack compaction invocations (n) and cycle counts per benchmark.

Benchmark	**Cycles**	**n**
129.compress	2.68B	4
164.gzip	24.5B	1
176.gcc	3.64B	2
181.mcf	591M	0
197.parser	2.44B	2
253.perlbmk	9.76B	4
256.bzip2	1.31B	0

According to Table 5.1, this procedure is required at most four times in the cases of 129.compress and 253.perlbmk, with it not being required at all in the cases of 181.mcf and 256.bzip2. Compacting the stack between zero and four times at a penalty of 488 000 cycles each is extremely insignificant with respect to these benchmarks' dynamic cycle counts also shown in Table 5.1. This is in addition to the fact that 16 active frames at 4-KB each is viewed to be very conservative with respect to the average case. Therefore, it is not unreasonable for the effect of stack compaction to be considered negligible in the following performance measurements.

5.2 Benefits

Having investigated the overhead associated with this new allocation scheme, the following sections are meant to showcase its abilities. We will show that this scheme does what was originally intended, as well as demonstrate potential performance gains even with a limited simulation environment.

5.2.1 Antidependences

First and foremost, this new stack frame allocation scheme is designed to rename stack frames, and therefore eliminate a common antidependence between the loads and stores of sibling procedures. To determine if it accomplishes this goal, the stack antidependence study from Section 2.1 was repeated on the benchmarks recompiled with the freelist-based allocating compiler. As illustrated in Table 5.2, the overall amount of antidependences on the stack with a distance less than 100 instructions dropped dramatically. In four of the seven benchmarks, the number of these antidependences in the case of the freelist-based allocating compiler dropped to below 5% of those in the stack-based case. Yet for the remaining benchmarks this amount varied widely, as in the case of 256.bzip2, which retained over 50% of its stack-based antidependences.

Table 5.2: Number of stack stores with antidependences less than 100 instructions per benchmark for stack-based and freelist-based frame allocation. The average decrease in these dependences from the stack-based to freelist-based case is 85.1%.

Benchmark	**Stack-Based**	**Freelist-Based**	**% Dec.**
129.compress	23 315 128	5 890 043	74.7
164.gzip	14 760 590	214 433	98.5
176.gcc	23 930 119	3 749 079	84.3
181.mcf	711 383	20 211	97.2
197.parser	9 609 401	409 731	95.7
253.perlbmk	22 359 295	22 400	99.9
256.bzip2	4 252 667	2 325 544	45.3

In addition to decreasing the total amount of stack antidependences less than 100 instructions, the distance profile also changes dramatically in the freelist-based allocation case. The overall amount of antidependences with distances less than 10 instructions actually increases in all cases but one (176.gcc). In addition, these antidependences now comprise the vast majority of those with distances less than 100 instructions for the freelist-based allocation case, as shown in Figure 5.2

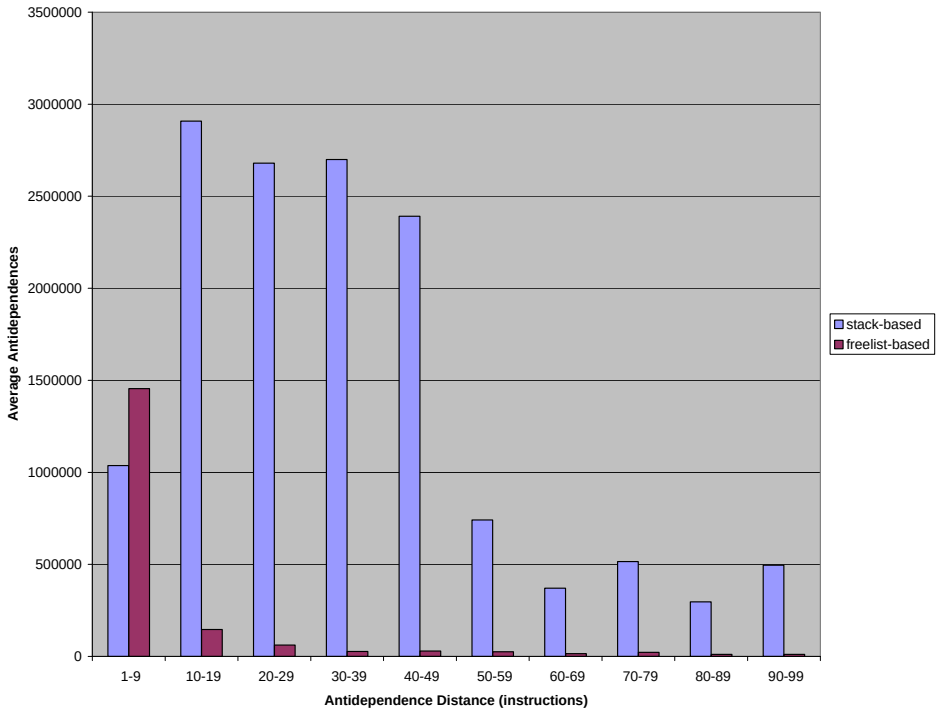


Figure 5.2: Histogram of stack stores with antidependences less than 100 instructions averaged over the seven SPECint benchmarks for stack-based and freelist-based allocating compilers. Stack stores with antidependence distances less than 100 instructions comprise only 3.5% of all program stores in the freelist-based allocation case compared to the 27% of Figure 2.1 in the stack-based allocation case.

and as compared with Figure 2.1. The red bars in Figure 5.2 are believed to represent those antidependences on the stack unrelated to frame allocation.

Given their increase both in number and in prominence in freelist-based allocation, stack stores with antidependence distances less than 10 instructions are likely to be caused by something other than stack frame allocation. These antidependences may be the result of a processing loop where an operand is loaded from the stack, processed, then stored back. Another possible scenario is that the antidependence is loop-carried; for example, in each iteration of a loop, a structure is allocated, operands are stored to it, then later reloaded for processing. The latter is believed to be the case in 129.compress, which has the highest percentage of its stack stores with antidependences of distance less than 100 instructions (97.4%) fall into the distance less than 10 instructions category. As for why the dependences in this category increase over almost all the benchmarks, this is believed to be caused by the register spilling problem discussed in Section 5.1.1. Register spilling in a tight

loop would cause an extra loop-carried antidependence of small distance. Again, 129.compress is the most affected by this, with its stack stores with antidependences of distance less than 10 instructions increasing from 2 762 258 in the stack-based allocation case to 5 736 810 in the freelist-based allocation case.

By all accounts, freelist-based stack frame allocation achieves a significant reduction in stack antidependences with distances less than 100 instructions, which are encountered empirically in roughly one-quarter of all store instructions for traditionally compiled programs. This is believed to be significant in potential, perhaps as showcased in the section below.

5.2.2 Performance

So far it has been shown that freelist-based allocation carries an overhead, but also is successful in significantly reducing stack antidependences. Now it would be prudent to compare the scheme's performance with that of the traditionally compiled stack. This could be done directly with a performance simulator without particular regard to machine parameters other than ensuring realism, but that would not produce meaningful or interesting results. It has been said that this new allocation scheme would most likely benefit environments with increasingly aggressive front-ends; the more parallelism that is sought, the more stack frame allocation is brought into focus.

In order to (somewhat clumsily) model this, several enhancements were made to the base configuration of Section 4.2. Aggressive instruction supply schemes such as dynamic multithreading significantly reduce the effect of branch mispredictions on parallelism. Therefore, we start with a tournament-style predictor and only signal one-sixteenth of its actual mispredictions. Multithreaded architectures also usually have increased numbers of execution units to consume the parallelism made possible by the front-end. We therefore model a 32-wide machine, to eliminate resource constraints. Lastly, the fact that freelist-based frame allocation spreads out stack address utilization causes unsavory behavior in the L1 D-cache. Although intelligent cache designs could likely minimize this effect, cache design is beyond the scope of this thesis. To eliminate this effect in simulation, all accesses are assumed to hit in the L1 caches. Lastly, since the goal of this analysis is to determine the effect of stack frame renaming, a nonrenaming store queue was implemented. Instead of a load potentially receiving its value from multiple stores to the same address, it can only bypass from the latest store to that address. If the correct store from which a particular load should

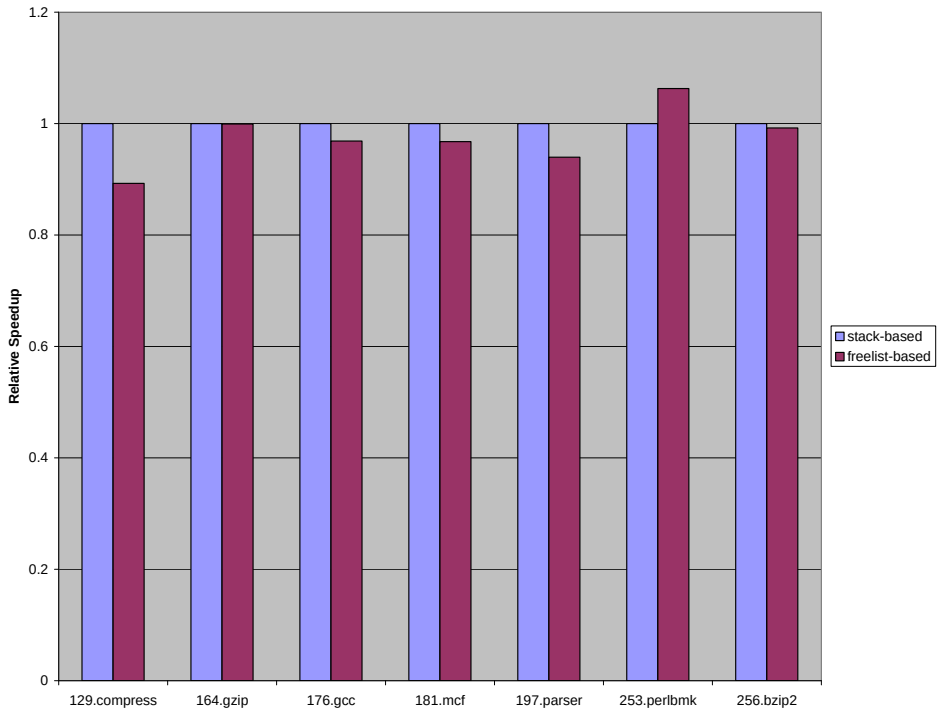


Figure 5.3: Relative performance of stack-based versus freelist-based allocating compilers. In addition to the base case of Table 4.1, this study assumes a 32-wide machine, a nonrenaming store queue, and enhanced branch prediction.

receive its value is not the latest store to that address and has not yet retired, a misspeculation is signaled and the load and all subsequent instructions are replayed.

With this configuration, all seven benchmarks were simulated with each compiler. Figure 5.3 illustrates the result in relative speedup. In four of the seven benchmarks, there is a somewhat significant loss of performance (average 6.3%) when considering a freelist-based stack allocation scheme. However, in two cases (164.gzip, 256.bzip2), the overhead introduced by the recompilation is overcome by the decrease in dependences. This is due to a decrease in misspeculations caused by the nonrenaming store queue. In the case of 253.perlbmk, freelist-based allocation actually outperforms stack-based allocation by 9%. After further investigation of this result, it was determined that although it is true 253.perlbmk suffers many more load misspeculations with stack-based allocation than with freelist-based, these misspeculations are non-frame-related and are only alleviated in the freelist case by the introduction of extra code which sufficiently separates the dependent instructions as to avoid the misspeculation.

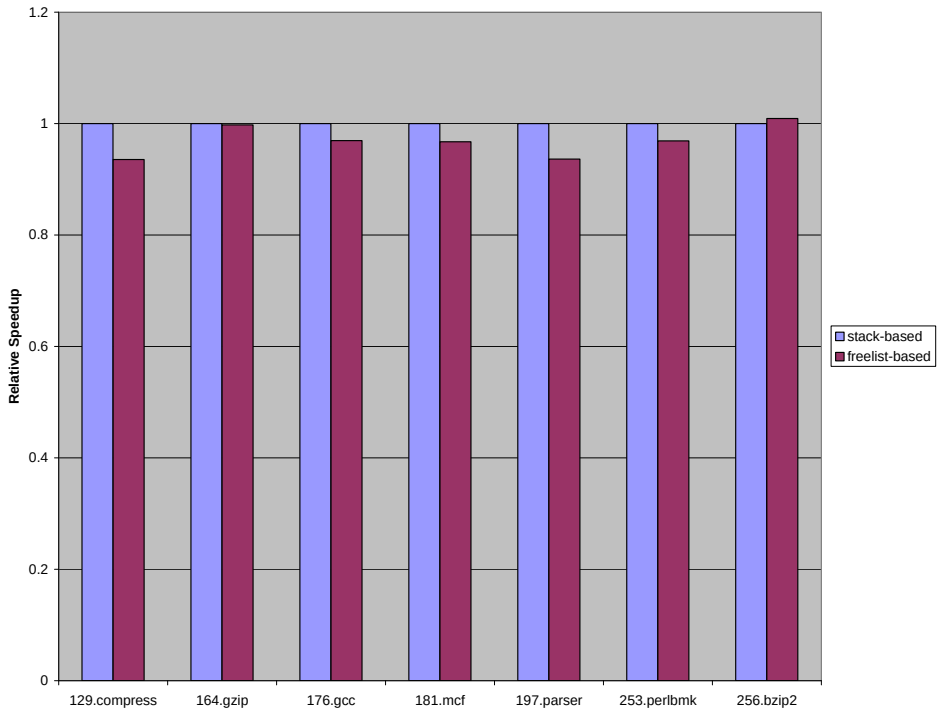


Figure 5.4: Relative performance of stack-based versus freelist-based allocating compilers. In addition to Figure 5.3, this study assumes memory dependence prediction with extension for non-renaming store queue and unconstrained issue width.

In order to further illuminate the effect of stack frame renaming, some additional changes were made to the environment. First, a Chrysos and Emer memory dependence predictor [8] was added to reduce "noisy" behaviors such as with 253.perlbmk described above. Now, after a load misspeculates with respect to its dependent store, a store set will be allocated for that load, and upon subsequent executions it will be forced to delay issue until its dependent store has issued. To accommodate the nonrenaming store queue in our model, an extension of the traditional dependence predictor was implemented. In this extension, stores which violate antidependences with loads are allocated a load set, and subsequently forced to delay issue. It was also determined that our issue model was somewhat limiting parallelism in the sense that if multiple instructions were vying for issue resources, preference was given to those occurring earlier in program order. This is undesirable when meaningful results depend on memory instructions issuing out-of-order. Therefore, in the following results, issue width is unconstrained.

Figure 5.4 shows the effects of memory dependence prediction and infinite issue width. Notice

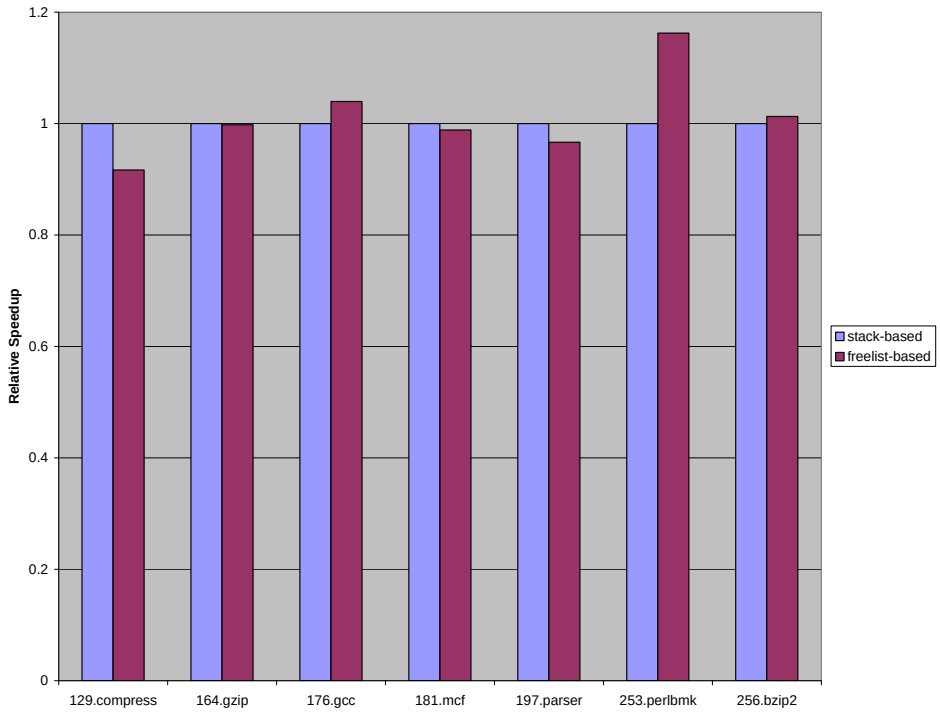


Figure 5.5: Relative performance of stack-based versus freelist-based allocating compilers. In addition to Figure 5.4, this study assumes perfect branch prediction.

that the speedup of 253.perlbmk in Figure 5.3 is now gone. This is due to the dependence predictor enforcing the store/load ordering that was only artificially enforced previously by the freelist-based allocating compiler due an increase in instruction count. Benchmarks 129.compress and 256.bzip2 show noticeable improvements, with freelist-based allocation outperforming stack-based in 256.bzip2. The rest of the benchmarks show little difference between Figure 5.3 and Figure 5.4.

In one last attempt to simulate as much parallelism as possible, perfect branch prediction was added to the model. With perfect branch prediction, antidependent stack memory instructions cannot be broken up by a pipeline flush associated with a mispredict. This is somewhat reasonable, since dynamic multithreading provides much of the same benefit; when one thread suffers a branch misprediction, the others can continue executing speculatively while the offending thread replays. Figure 5.5 shows the results with the addition of perfect branch prediction. Here, 176.gcc and 253.perlbmk join 256.bzip2 in cases where freelist-based frame allocation outperforms stack-based. Unlike before, in the case of Figure 5.5 the 16.3% speedup obtained by 253.perlbmk is not due to the noise that was eliminated by the addition of a dependence predictor. Instead, this particular

benchmark responds very well to the available parallelism enhanced by the reduction of antidependences on the stack. A closer look at the dependence prediction sets for the study of Figure 5.5 for all seven benchmarks is provided in Table 5.3. In the freelist-based allocating compiler for 253.perlbmk, only 637 731 stack stores were forced to issue late due to antidependences with prior loads, as opposed to 7 330 748 stores in the stack-based case. While this seems to correspond nicely with the performance gain seen for 253.perlbmk, other benchmarks are not all similarly correlated. There are likely other interfering behaviors in these cases we do not yet understand, which are either specific to a particular benchmark, or attributable to the limitations of the simulation environment. An investigation into these effects would represent a desirable direction for future work.

Table 5.3: Number of stack stores forced to issue late by the dependence predictor per benchmark for stack-based and freelist-based frame allocation. The average decrease from the stack-based to freelist-based case is 62.5%.

Benchmark	**Stack-Based**	**Freelist-Based**	**% Dec.**
129.compress	730 902	484 009	33.8
164.gzip	681 878	92 005	86.5
176.gcc	9 630 261	1 970 505	79.5
181.mcf	23 940	20 437	14.6
197.parser	6 754 689	202 707	97.0
253.perlbmk	7 330 748	637 731	91.3
256.bzip2	347 181	226 273	34.8

Overall, the results of these studies are viewed as promising. Multithreading provides just one way in which instruction supply mechanisms are becoming increasingly agile; freelist-based frame allocation helps relieve the pressure of renaming on execution resources created by increased parallelism. Furthermore, since stack-based allocation has been the enduring paradigm, there are likely other constructs (such as fully associative renaming load/store queues) designed to contend with the false dependences it produces. A re-evaluation of these constructs could lead to a reduction in design complexity.

CHAPTER 6

RELATED WORK

The concept of memory renaming dates back to the IBM System/360 Model 91 [9]. Traditionally, it is accomplished in hardware by an associative memory called the load/store queue. The purpose of this queue is threefold. First, it maintains store ordering until retirement for in-order update of the architectural state. Next, it provides store to load bypassing by caching the values of several in-flight stores for each memory location. Finally, the load/store queue is used for dependence validation, to ensure that correct program semantics were maintained. A good historical overview of these structures is given in [2] along with insight into their scalability in light of increasing instruction window sizes. Franklin and Sohi introduce the Address Resolution Buffer (ARB) in [7], which hashes memory references to different banks based on their addresses as an attempt to reduce the associativity required for a lookup. These two papers represent some of the recent attempts to reduce the pressure on the load/store queue resulting from higher ILP. In contrast, freelist-based stack frame allocation provides memory renaming in software, effectively reducing this pressure and perhaps allowing a re-evaluation of load/store queue design.

With respect to stack frame allocation, the costs of heap-allocated and stack-allocated activation frames were studied by Appel and Shao in [10]. This analysis was performed in the context of a compiler for the functional, garbage collecting language ML. Such languages with higher-order functions need closures to hold certain function variables. These closures do not conform to C-like scoping rules, and therefore cannot be allocated on the stack. Appel's argument is that, rather than allocating some procedure-support structures on the stack and some on the heap, it is simpler and just as cost-effective to put them all in the heap. Freelist-based stack frames are very similar to the heap frames described by Appel, and many of the costs he discusses are similar to what has

been discussed in this thesis. The difference is that freelist-based frames are still allocated on the stack and are not geared toward functional programming languages. Instead, freelist-based frames provide a reduction in false memory dependences through stack frame renaming.

CHAPTER 7

CONCLUSION

As instruction fetch mechanisms improve in their ability to provide larger instruction windows from which to extract parallelism, stack allocation considerations will become too important to ignore. This thesis contributes a compiler-directed, freelist-based, stack frame renaming mechanism that significantly reduces antidependences between stack memory instructions while efficiently managing stack space in the background. The reduction of stack antidependences addresses just one engineering problem in the larger scope of the aggressive pursuit of parallelism; designers will be forced to contend with many aging paradigms such as stack-based frame allocation in the near future.

REFERENCES

[1] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, December 1995.

[2] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable hardware memory disambiguation for high ILP processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2003, pp. 399–410.

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[4] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998, pp. 226–236.

[5] M. Franklin, "The multiscalar architecture," Computer Sciences Department, University of Wisconsin - Madison, Tech. Rep. 1196, November 1993.

[6] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 414–425.

[7] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[8] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 142–153.

[9] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina, and J. W. Smith, "The IBM system/360 model 91: Storage system," *IBM Journal of Research and Development*, vol. 11, pp. 54–68, January 1967.

[10] A. W. Appel and Z. Shao, "An empirical and analytic study of stack vs. heap cost for languages with closures," *Journal of Functional Programming*, vol. 6, no. 1, pp. 47–74, Jan 1996.

APPENDIX - STACK COMPACTOR

The following listing is an implementation of the stack compactor described in Section 3.2.2. This particular implementation is written in C, and assumes that the stack frame that caused compaction is preallocated and its size not yet stored.

```
unsigned* stack_compactor(unsigned* fp, unsigned last_frame_size) {
   unsigned* from;
   unsigned* to;
   unsigned* rlist;
   unsigned* temp;
   unsigned* forward;
   unsigned* next_forward;
   unsigned frame_size;
   unsigned i;

   /* reverse the linked list */
   temp = fp;
   forward = (unsigned*)*temp;
   *temp = (unsigned)0;
   next_forward = (unsigned*)*forward;
   (unsigned*)*forward = temp;
   temp = forward;
   while (1) {
      if (next_forward == (unsigned)0)
         break;
      forward = next_forward;
      next_forward = (unsigned*)*forward;
      (unsigned*)*forward = temp;
      temp = forward;
   }
   rlist = temp;

   /* compact the stack */
   from = (unsigned*)*rlist;
   frame_size = *(from + 1);
   to = (unsigned*)((unsigned)rlist - frame_size);
   temp = rlist;
   while (1) {
      if (from != to) {
         for (i = 0; i < frame_size/4; i = i + 1) {
            *(to + i) = *(from + i);
         }
      }
      (unsigned*)*temp = to;
      temp = to;
      if (*from == (unsigned)0)
         break;
```

```
        from = (unsigned*)*from;
        frame_size = *(from + 1);
        if (frame_size == (unsigned)0)
           frame_size = last_frame_size;
        to = (unsigned*)((unsigned)to - frame_size);
     }

     /* re-reverse (restore) linked list */
     temp = rlist;
     forward = (unsigned*)*temp;
     *temp = (unsigned)0;
     next_forward = (unsigned*)*forward;
     (unsigned*)*forward = temp;
     temp = forward;
     while (1) {
        if (next_forward == (unsigned)0)
           break;
        forward = next_forward;
        next_forward = (unsigned*)*forward;
        (unsigned*)*forward = temp;
        temp = forward;
     }
     fp = temp;
     return fp;
  }
```