Toward An Optimizing JIT Compiler For IA-64

Samuel Sanseri

June 2001

Abstract

This project lays the groundwork for porting an existing Java JIT compiler to the IA-64 architecture. There are three main tasks involved in this port, all of which involve only the back-end (machine-dependent portion) of the existing compiler: (1) efficient instruction packing of IA-64 native code into the IA-64 long instruction word, (2) implementing a dynamic method-translation subsystem (*trampolines*) for the IA-64 processor, and (3) implementing translation of compiler intermediate instructions into native IA-64 instructions. We have documented all three tasks, and have implemented IA-64 instruction packing using state-of-the-art search techniques, obtaining optimization competitive with a recently-released IA-64 C compiler.

Chapter 1

Introduction

JIT compilation for IA-64 differs from static compilation for RISC or CISC architectures in two respects: (1) the IA-64 architecture vs. traditional RISC or CISC architectures and (2) JIT compilation vs. whole-module compilation. Because of these differences, porting an existing JIT compiler to IA-64 poses some unique and interesting challenges.

1.1 Instruction Packing: IA-64 vs. Traditional Architectures

The IA-64 architecture and the recent RISC architectures differ in resource scarcities. Traditional RISC architectures (and CISC architectures even more so) suffer from a lack of registers, and so a whole area of research has sprung up on compiler register allocation for these architectures. The IA-64 architecture has dealt with this problem by putting 128 general-purpose registers on its processors. Compared to compilers for the SPARC [14] or i386 [8] processors, the compilers for IA-64 processors have a gold mine of registers to work with, so running out of registers will probably not be a problem on this architecture. However, IA-64 has a unique challenge that these other architectures do not face, namely the problem of packing instructions into its long instruction word in such a way that maximum parallelism and throughput can be achieved.

1.2 Instruction Packing: JIT vs. Whole Module Compilation

Another aspect that makes our work interesting is the focus on real-time optimization (i.e. time spent compiling "counts"). Optimizing just-in-time (JIT) compilers face a challenge that traditional whole-module compilers do not: The time used for optimization by whole-module compilers is not included in the end user's runtime, but JIT compilation time is born as an additional cost to the

end user. Our instruction-packing techniques are well-suited to JIT compilers as well as to whole-module compilers because they are *anytime*, i.e. the longer they are allowed to run, the better results they are expected to achieve.

1.3 Porting an Existing JIT Compiler: Trampolines And icode Instructions

The goal in any port of an existing compiler should be to accelerate development time by reusing existing code and design ideas. The Kaffe [12] Java JIT compiler has an good record of successful ports to multiple architectures, but it has not yet been ported to IA-64. We view this as a worthwhile goal. In a port of Kaffe to any architecture, there is architecture-independent code which can be reused and architecture-dependent code which must be custom implemented for the target architecture. The architecture-independent part can mostly be ignored by the porting developer, so we will not consider it here. The architecture-dependent portion of Kaffe contains two pieces, the trampoline subsystem for method compilation, and an intermediate *icode* between Java bytecode and native machine code, which must be translated to native code for the target architecture.

1.4 Overview

Chapter 2 describes previous work in the areas of IA-64 compilation, instruction packing, register allocation, method-compilation techniques, and Kaffe's icode system. The most important contribution of our work is our instruction packing technique. Informal and formal descriptions of the IA-64 architecture and the instruction packing problem model are given in Chapter 3. Chapter 4 gives details of the implementations, search spaces, and experimental results. Chapter 5 documents the requirements for implementing a trampoline subsystem. Chapter 6 suggests future work and Chapter 7 summarizes the project.

Chapter 2

Previous Work

Creating a JIT compiler for IA-64 involves sensible synthesis and extension of previous work in the areas of IA-64 compilation, instruction packing, register allocation, trampolines, and icode instructions. This chapter describes previous work in these areas.

2.1 IA-64 Compilation

The IA-64 is a relatively new architecture, and not many compilers for it have been released to the general public. We spent a significant amount of time examining the Trimaran C compiler and research infrastructure [4] for an IA-64-like architecture. However, we were unable to get this infrastructure program to compile and run on our RedHat Linux system. The first publicly available C compiler for IA-64 on Linux was released by Cygnus [18] in February 2000. An optimizing C compiler was released by SGI [17] in May 2000, too late to be included in this report. We are not aware of the existence of any optimizing Java JIT compilers for IA-64 at this time, but we have been told that there are efforts in progress (for example at Cygnus). Our results in Chapter 4 will be compared with the Cygnus GNU C compiler (throughout this paper we call this compiler "GCC").

2.2 Instruction Packing

GCC performs a simple form of heuristically-guided instruction packing with no search for more efficient packings. We are not aware whether any other compilers perform more sophisticated instruction packing, but we suspect that the recently-released SGI optimizing C compiler will outperform GCC. We hope to investigate further soon.

2.3 Register Allocation

Although register allocation is not a serious problem for the IA-64 architecture, it does have interesting challenges that are similar to those of instruction packing. Instruction packing is a form of slot allocation, where the slot is the scarce resource that needs to be efficiently allocated to instructions, and instructions may be reordered in legal ways to better fill the available slots. Analogously, register allocation is a problem where registers are the scarce resources that must be efficiently but correctly allocated to instructions, and different registers may be used based on source or intermediate code semantics. Both problems can be viewed as a search or scheduling problem. Some powerful register allocation techniques have been recently developed, for example, hierarchical cyclic interval graphs [3].

2.4 Trampolines

Powerful techniques for JIT compilation have also been announced recently. For example, IBM has an article describing optimization techniques used in their Java Virtual Machine (JVM) [19]. These techniques should be explored by developers who are either implementing an entire JVM or optimizing an existing JVM. However, our goal was to lay the groundwork for porting the Kaffe Java JIT compiler to IA-64; we did not explore the techniques in these papers, which mostly dealt with areas in which Kaffe had already made architecture-independent design decisions. However, the trampoline subsystem, which must be implemented when porting Kaffe to a new architecture, interested us. The Kaffe web page mentioned that documentation for porting Kaffe to a new architecture would be helpful. Although they had a document attempting to explain trampolines [5], it was difficult to read, and it took us a significant amount of time before we understood this subsystem. Therefore, we documented the requirements for implementing trampolines in Kaffe, along with case studies from existing Kaffe implementations. Portions of our trampoline documentation [16] are included in Chapter 5.

2.5 icode Instructions

Another required portion of a port to a new architecture is a translation of Kaffe's icode instructions to native code. Kaffe uses icode instructions as an intermediate level between Java bytecode and architecture-specific machine code to make their JIT compiler more portable. We created a table [15] based on existing Kaffe implementations describing the requirements for translating each icode instruction to native code as well as the usage of most of the icode instructions.

Chapter 3

IA-64 Instruction Packing for Basic Blocks

Some sort of instruction packing must be implemented for a JIT to run on IA-64. An efficient instruction packing technique is preferred. Efficiency of an instruction packing technique for IA-64 is measured in two respects: how long it takes to schedule the instructions and how long the resulting schedule takes to run.

This chapter describes the IA-64 instruction packing problem. First the instruction packing problem and the IA-64 architecture [6, 10, 7, 11] are informally described in Sections 3.1 and 3.2, respectively. Then a formal problem model is given in Section 3.3. Finally, the IA-64 architecture is presented as a partial instantiation of this formal problem model in Section 3.4. Chapter 4 describes how efficiency is achieved in our implementation.

3.1 An Informal Problem Description

The formal model of the IA-64 instruction packing problem is difficult to understand without an informal description. The instruction packing problem is a type of *resource-constrained project scheduling* problem (or a *scheduling* problem for short). Scheduling problems involve assigning times to tasks in ways that do not violate given sets of precedence constraints and resource constraints, while at the same time finding an optimal value for some objective function which characterizes the "goodness" of the schedule.

In the instruction-packing problem, the tasks are instructions and the times are positions in the instruction output stream. The resource constraints involve the processor's available execution units, the available combinations of instruction types which can be combined in the processor's long instruction word, and the registers and memory on the processor. Precedence constraints between instructions are computed based on the register and/or memory resources used by the instructions. The scheduling algorithm is allowed to reorder instructions

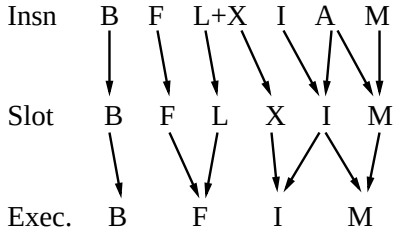

Figure 3.1: Mappings from instruction types to slot types and from slot types to execution unit types.

in any way that does not violate these constraints. The objective function is an estimate of a penalty proportional to the cycle time required for the processor to execute the given instruction schedule.

3.2 An Informal Description Of The IA-64 Architecture

The IA-64 computer architecture uses a *bundle* or long instruction word (LIW) which holds three instruction slots. Each bundle is 128 bits long: 41 bits for each of the three instruction slots and 5 bits for a template.

Only certain combinations of instruction types can be packed into a bundle. The allowed combinations are specified by *templates*. Templates for the IA-64 long instruction word specify the types of the three instruction slots that will be used by the corresponding instructions in the bundle. There are twenty-four allowed templates.

There are six *instruction types*: A, M, I, F, B, and L+X. Type A is for integer ALU instructions, type M for memory load and store instructions, type I for integer non-ALU instructions, type F for floating-point instructions, type B for branch instructions, and type L+X for long integer instructions. Most instructions fit in a single slot. Only type L+X instructions require two slots.

There are six *template instruction slot types*: M, I, F, B, L, and X. Figure 3.1 shows how instruction types match slot types. Type A instructions *match* (can be put in) either type M or type I instruction slots. Type M instructions match only type M slots, likewise type I instructions match only type I slots, type F instructions match only type F slots, type B instructions match only type B slots, and type L+X instructions (which use two slots) match a pair of slots (one type L slot and one type X slot).

There are four *execution unit types*: M, I, F, and B. The IA-64 processor can execute at most six instructions simultaneously, subject to certain resource and precedence constraints. The processor has multiple execution units of each type. It has two type M execution units, two type I execution units, two type F execution units, and three type B execution units. Each of these units can execute at most one instruction per cycle. Type M slots will always execute

on type M execution units, type I slots will always execute on type I execution units, type F slots will always execute on type F execution units, and type B slots will always execute on type B execution units. The type L/type X slot combination for the type L+X instruction will execute such that the type L slot executes on a type F execution unit at the same time as the type X slot executes on a type I execution unit; these two portions of the same instruction must execute in the same cycle.

Templates also specify *stops* between instruction slots in a bundle or between the last slot of a bundle and the first slot of the next bundle. A stop is a command from the compiler to the hardware to delay at least one cycle between execution of instructions before the stop and instructions after the stop, in order to copy all the updated registers to all execution units that can use them. The compiler must use templates that contain stops in the following two circumstances: (1) between any two (not necessarily consecutive) instructions that both write to the same register (*write-after-write* or *WAW*) and (2) between any two instructions in which the earlier instruction writes to a register that is read by the later instruction (*read-after-write* or *RAW*). However, a stop is not required between two instructions in which the earlier instruction reads from a register and the later instruction writes to the same register (*write-after-read* or *WAR*). These requirements for placement of stops apply only to register dependencies; they do not apply to memory dependencies.

A compiler for the IA-64 architecture must ensure two things in order to guarantee correctness of the parallelized code: (1) that hardware hazards are avoided and (2) that semantics of the code is the same as if the original instruction stream were executed in order, one instruction at a time. The IA-64 architecture provides the compiler with a specific mechanism called an *instruction group* for ensuring these two conditions in the parallelized code. An instruction group is an ordered list of parallel instructions beginning with an instruction that immediately follows a stop and ending with an instruction that immediately precedes a stop.

A write-after-write register dependency inside an instruction group is a hardware hazard, which would result in a hardware exception on IA-64. However, a read-after-write register dependency inside an instruction group is not a hardware hazard; it is a semantic error which cannot be detected by the hardware. It cannot be detected because the definition of instruction groups allows write-after-read conditions between instructions in the same instruction group to be reordered. For example, the original instruction sequence containing $insn_i$ (read r), $insn_i + 1$ (write r) could be reordered in the parallelized form as $stop, \ldots, insn_i + 1(writer), insn_i(readr), \ldots, stop$, with the same semantics as the original sequence. Therefore the compiler must ensure semantic correctness by placing stops between all read-after-write register dependencies, and it must ensure hazard elimination by placing stops between all write-after-write register dependencies.

Dependencies between loads and stores of memory locations are different from register dependencies on the IA-64 architecture. The ordering of two store instructions or of one load instruction and one store instruction must be pre-

served in the parallelized code if the memory addresses overlap, but these instructions can legally occur in the same instruction group. The IA-64 L1-cache will correctly handle multiple loads and stores within the same instruction group. Since copies of the cache are not maintained within the execution units, the L1-cache results are available to subsequent instructions without requiring a stop between them, but the processor will automatically delay two cycles between a load from memory and a use of the result from memory. [11]

A branch instruction must be the last instruction in its basic block. It cannot be reordered in the parallelized code. If there are any read-after-write constraints between previous instructions and the branch instruction, they are an exception to the rule about read-after-write dependencies, and may be placed in the same instruction group as the branch instruction.

3.3 The Formal Problem Model[1]

PROBLEM: Basic Block Instruction Packing (Decision Problem)

INSTANCE: A Basic Block Instruction Packing instance consists of two parts: an architecture description consisting of

- A set T_I of instruction types, including a distinguished element B.
- A set T_P of template (pattern) types.
- A set $L = R \cup \{m\}$ of locations (registers and memory) accessed by instructions, where R is a set of registers. Let $N_R = |R|$ and $N_L = |L| = N_R + 1$.
- A set of bundle templates (patterns) P, each a sequence of template elements drawn from the set $T_P \cup \{S\}$, where S represents a "stop".
- A relation $\triangleright \subseteq T_I \times T_P$ which holds iff a given instruction type from T_I matches a given template type from T_P.
- An objective function $f : \text{seq}\, P \rightarrow N$ mapping a sequence of bundle template to a "penalty" for the sequence.

and target data consisting of

- A sequence I of instructions, each of the form $\langle t, Reads, Writes \rangle$ where
  - $t \in T_I$ is the type of the instruction.
  - $Reads, Writes \subseteq L$ are the locations respectively read and written by the instruction.

  Only the last instruction in the sequence can be of type B.
- An integer quality bound q on the value of the objective function.

[1] Thanks to Bart Massey for help with the original version of this section.

DEFINITIONS: A number of definitions will be useful in posing the question below.

Define the following relations on any instruction sequence I for constraints between two instructions $i, j \in I \quad (i \neq j)$:

- Let $i \prec_G j$ denote a *register dependency* between instructions i and j: either a value is read by j from register r after it was placed there by i (RAW), or a value is clobbered by j after it was written by i to register r (WAW). This means that i must be scheduled in an earlier group than j.
- Let $i \prec_S j$ denote a *memory or branch dependency* between instructions i and j. This means that i must be scheduled in an earlier slot than j.
- Let the *order dependency* relation $\prec_{S,G}$ be the transitive closure of $\prec_G \cup \prec_S$.
- Let $i \preceq_G j$ denote a *WAR register dependency* between instructions i and j: a value in register r is clobbered by j after being read by i (WAR). This means that i must be scheduled either in the same group as j or in an earlier group than j.

A bundle template P *matches* a sequence I of n instructions iff there exists a reordered, lengthened, delayed, and stopped version C of I

$$C \;=\; (S \circ X \circ N \circ \pi)(I)$$

with the properties that

1. The size of the template P equals the size of the modified instruction sequence C.
2. Instruction types or stops in C match the template types or stops in P using the given *match* relation.

In other words,

$$|C| = |P| = n$$
$$\forall k \in \{1, \ldots, n\} \;.\; (C_k = P_k = S) \;\vee\; (t(C_k) \;\triangleright\; t(P_k))$$

For any instruction sequence I

- A *reordered sequence* $\pi(I)$ is obtained by reordering elements of I in any manner that does not violate the ordering requirement of $\prec_{S,G}$.
- A *lengthened sequence* $N(I)$ is obtained by inserting any number of *typed no-op* instructions $\langle t, \emptyset, \emptyset \rangle$ into I, where t is an arbitrary type from T_I.

- A *delayed sequence* $X(I)$ is obtained by inserting some number of stops S between adjacent instructions of I in any manner that does not violate the grouping requirement of $\preceq_G$ (i.e. if $i < j$ and $i \preceq_G j$ and π reordered i and j such that j comes before i, then no stop can legally be placed between j and i).
- A *stopped sequence* $S(I)$ is constructed by inserting stops S between pairs of (not necessarily consecutive) instructions i and j of I in such a way that all register dependencies are *broken* (i.e. there will be a stop somewhere between i and j if $i \prec_G j$), without violating the grouping requirement of $\preceq_G$(i.e. if $i < j$ and $i \preceq_G j$ and π reordered i and j such that j comes before i, then no stop can legally be placed between j and i. For some sequences it may not be possible to satisfy both requirements.)

Extending this notion, a sequence B of n bundle templates *matches* a sequence I of instructions iff there is a partition of I into n contiguous subsequences $I_1 \ldots I_n$ such that each template B_k matches the corresponding subsequence I_k.

QUESTION: Does there exist a sequence B of bundle templates which matches the given sequence I of instructions, such that $f(B) < q$?

Several things are worthy of note in this description:

1. A *basic block* is a sequence of instructions with at most one branch, which can only occur at the end of the block. This is the inspiration for the treatment of the B instruction type in the problem description.
2. The IA-64 allows *predicated execution*: we assume all predicates are true, to simplify the search. Cleverer predicate analysis should be easy to accomodate in the problem description.
3. We assume that each register in the architecture has a unique, constant global name. For the IA-64, we can usually track register names across *window change* instructions in a prepass, and replace them with physical register names. However, these name changes are not handled by our implementation.
4. Any memory access is conservatively assumed to overlap with any other memory access, so there is only one memory location in L.

3.4 The IA-64 Architecture As A Partial Instantiation Of The Instruction Packing Problem

The details for the problem model of the IA-64 architecture are given in this section.

- A set T_I of instruction slot types, including a distinguished element B: $T_I = \{A, M, I, F, B, L, X\}$
- A set T_P of template (pattern) types: $T_P = \{M, I, F, B, L, X\}$
- A set $L = R \cup \{m\}$ of locations (registers and memory) accessed by instructions, where R is a set of registers: R = {r0,. . . ,r127, f0,. . . ,f127, p0,. . . ,p63, b0,. . . ,b7, ar.pfs}
- A set of bundle templates (patterns) P, each a sequence of template elements drawn from the set $T_P \cup \{S\}$, where S represents a "stop": P = {MII, MIIS, MISI, MISIS, MLX, MLXS, MMI, MMIS, MSMI, MSMIS, MFI, MFIS, MMF, MMFS, MIB, MIBS, MBB, MBBS, BBB, BBBS, MMB, MMBS, MFB, MFBS}
- A relation $\triangleright \subseteq T_I \times T_P$ which holds iff a given instruction type from T_I matches a given template type [2] from T_P: $\triangleright$= {A $\triangleright$ M, A $\triangleright$ I, M $\triangleright$ M, I $\triangleright$ I, F $\triangleright$ F, B $\triangleright$ B, L $\triangleright$ L, X $\triangleright$ X }
- An objective function $f : \text{seq}\, P \rightarrow N$ mapping a sequence of bundle templates to a "penalty" for the sequence: $f : \text{seq}\, P \rightarrow N = 6 * (\text{numStops}) + (\text{numNops})$, where *numStops* is the number of times a stop S appears in the sequence and *numNops* is the number of positions in the sequence which contain a typed no-op instruction. [3]

[2] In IA-64, the type L+X instruction is a two-slot instruction which requires a pair of consecutive slots, the first being a type L slot and the second a type X slot. This is not a problem for this relation because only two templates are available for the type L instruction (MLX and MLXS), and both of them have L immediately followed by X. Since no other instruction type matches with template types L or X, a type L+X instruction can be matched by matching the first slot of the instruction with a template type L, and the second slot of the instruction with the template type X; if the first part of the instruction matches, the second part always will.

[3] An exact cycle count using information from the IA-64 Microarchitecture Reference Manual [11] would be more accurate; the given objective function is a first approximation of the "penalty" incurred by a schedule.

Chapter 4

Implementation

Chapter 3 described the instruction packing problem. This chapter explains how we implemented two different solutions to this problem and compares our experimental results with GCC. Our implementation uses two systematic search spaces, which are presented in Section 4.1. Section 4.2 discusses the details of the various modules in our implementation. Section 4.3 compares the results of our implementation with GCC.

4.1 Two Systematic Search Spaces (insnpack2 And insnpack3)

We implemented two different systematic search spaces called insnpack2 and insnpack3. The search algorithm for insnpack3 is given in Section 4.2. We now describe the search spaces involved. Section 4.1 shows the results for both search spaces and compares them with results for GCC. Both search spaces involve a layered search because two decisions may be required before the search routine can be recursively called.

4.1.1 The insnpack2 Search Space

The insnpack2 search space is finite and incomplete. In this search space, at each node of the search the schedule is systematically extended by computing the list of instructions whose parents have all been scheduled, and selecting one instruction from this list. Then insnpack2 scans the current bundle list to find a position where this selected instruction may be legally placed. If no legal position is found, then a template is selected and an empty bundle with this template is added to the end of the schedule. Then this instruction is inserted in the newly added bundle and the search routine is recursively called. Otherwise, if a legal position is found, then the instruction is simply inserted in the legal position, and the search routine is recursively called.

The insnpack2 search space for templates uses all 24 templates from the IA-64 specification. In the implementation for this search space, bundles do not have the ability to turn stops on or off, and the search routine does not insert stops. We note several things of interest about the insnpack2 search space and our implementation.

- It is not complete for our problem model because it requires a conservative implementation of WAR dependencies, so that a lower numbered instruction must always precede a higher numbered instruction if there is a WAR dependency between them.
- The layered search involves picking an insn first, then a template if necessary.
- Instructions can "back-fill" earlier slots in the schedule after later slots have been filled, due to the existence of instruction types. This makes it more difficult (though still possible) to calculate the number of nops in a partial schedule, which is important for pruning provably suboptimal schedules.
- There are 24 templates, bundles cannot turn stops in a template on or off, and the search routine does not insert stops.
- Instruction groups are created accidentally because of the way templates are chosen.
- Branch-and-bound pruning is not working as well as it could be. Because it was more difficult to count nops in this search space, we underestimated the number of nops until a complete schedule was reached. Also we were not calculating the number of instruction groups in this version of the program. Thus, we missed a lot of pruning opportunities which could have arisen each time a template was placed, which would cause the schedule size to exceed the required number of stops. By shrinking the search space in this way without pruning optimal solutions away, we could have found better solutions or the same solutions more quickly.
- Template-selection heuristics need improvement. We spent signficant effort on improving this for insnpack3, and it helped quite a bit.
- Limited-discrepancy search was not easily implemented for this search space. Because search space is layered so that an instruction is chosen before a template is chosen, there are times when there is no legal extension of the current schedule, and often this is the heuristic choice. Taking every heuristic choice as in LDS0 will often not result in arriving at a leaf node. This means that the search space must be modified before LDS can be used.

  The search space was originally implemented so that an instruction would be chosen based on the earliest time it could be scheduled based on precedence constraints, but ignoring grouping constraints. This is what causes

a heuristic choice to sometimes lead to a dead end. If grouping constraints are considered in the heuristic choice of instruction, LDS0 will contain a legal schedule, but this is a modification of the insnpack2 search space (as it was originally implemented). Preliminary testing showed that this modified search space produced worse results using LDS than the unmodified insnpack2 search space produced using DFS. However, these results could be a side-effect of the lack of a decent template-selection heuristic.

4.1.2 The insnpack3 Search Space

The insnpack3 search space is finite and incomplete. In the search routine for this search space, the current instruction group is checked to see if it is empty. If it is empty, then the next instruction group is obtained by selecting all instructions which have not been scheduled whose parents either have already been scheduled or have not been scheduled but the precedence between them is not of type R.

In this search space, at each depth of the search the schedule is systematically extended by first determining whether a new bundle is needed before an instruction can be inserted at the current time (slot location). If a bundle is needed, a template is selected and a bundle with this template is added to the end of the schedule. Then one of the instructions from the current instruction group which match this slot is selected and the search routine is recursively called. If no bundle is needed, then an instruction which matches the current slot is selected. This instruction is removed from the instruction group; if it was the last instruction in the instruction group, a stop is placed at the earliest possible time after this time. Then the search routine is recursively called with the depth and time incremented. If no instruction from the current instruction group matches this slot, then a nop is implicitly placed in this slot, the search is called with the same depth but with the time (slot position) incremented. (Note that depth simply defines how many instructions have been scheduled so far.) We note several things of interest about the insnpack3 search space.

- It is not complete for our problem model, because it greedily moves some instructions into an earlier instruction group than required. Later instruction groups may have extra nop "holes".
- The layered search involves picking a template (if necessary) first, then an instruction.
- Instructions cannot "back-fill" earlier slots in the schedule after later slots have been filled. This makes it easier to calculate the number of nops in the schedule for branch-and-bound pruning.
- There are only 10 templates; bundles can turn stops on or off; and the search routine does insert stops. This shrinks the search space.
- Instruction groups are chosen greedily; stops are only inserted when necessary to terminate an instruction group.

- WAR precedence types are fully supported. Thus a lower-numbered instruction is allowed to go after a higher-numbered instruction as long as there is no stop between them. This improved results over insnpack2; in some cases this results in fewer instruction groups in the output schedule.
- Branch-and-bound pruning greatly reduces the search space. There is room for still more pruning in a special kind of instruction group. If there are no type M precedences between instructions in the current instruction group, then the instructions in the current group can be scheduled in any order under the current model.[1] In this case, we only need to try one instruction for each instruction type that matches the current slot type. Pruning all other choices for instructions significantly reduces the branching factor.
- Instruction-selection heuristics need improvement. The insnpack3 implementation should bias toward selecting memory instructions.
- Limited discrepancy search is easily implemented for this search space and improves on the results using depth-first search.

4.2 The insnpack Implementation

The current *insnpack* implementation uses a partially automatic multi-step process as depicted in Figure 4.1. First, GCC is run on a C program to generate IA-64 assembly code. This assembly code is processed by an AWK preprocessor module with basic blocks tagged. The preprocessor output is broken into basic blocks by hand and each basic block is put into a separate file where the first line contains the number of instructions in the given basic block and the subsequent lines contain the instructions in the basic block. Finally, each basic block is fed individually to the main *insnpack* program along with a resource description file and a template description file. The program computes its resulting best schedule for the given basic block and prints it to the screen.

To compare these results with GCC, the C program is recompiled with debugging symbols on and GDB is run on the resulting a.out file. From GDB, the *main*() function is disassembled and the IA-64 assembly code is cut-and-pasted into a file. Then the basic block is located in the file, and the number of stops and nops in the given basic block is counted.

The rest of this section describes the architecture of the basic-block instruction packing program. We begin with a list of source files with line count statistics. Then we describe of each module, including information about what functions the code performs, and enough insight into tricky or unusual algorithms that our results can be recreated.

[1] This prune will not be possible in the model that handles execution unit types with delays between different instruction types

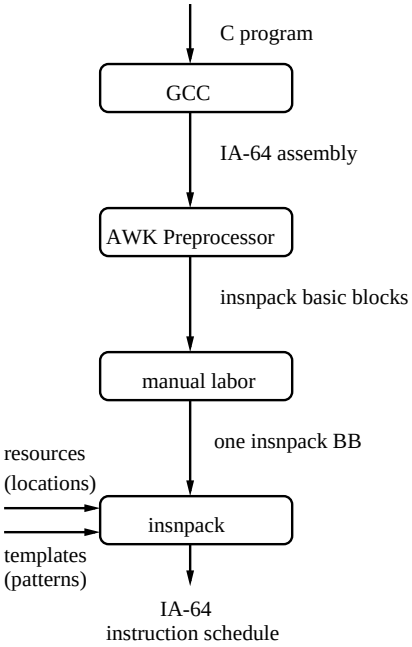

Figure 4.1: Steps to run insnpack

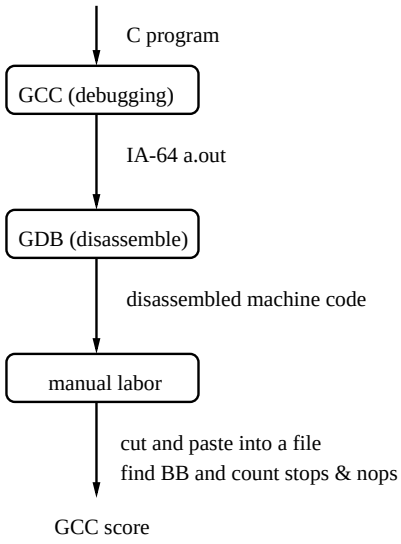

Figure 4.2: Comparing insnpack results with GCC

- Files: The Basic Block Instruction Packing program consists of the following thirteen files consisting of 3743 lines of code and data (line counts are from insnpack3).
  - insntrans.awk (457 lines)
  - Options.java (62 lines)
  - resource.txt (331 lines)
  - template.txt (12 lines)
  - insnpack.java (471 lines)
  - Resource.java (123 lines)
  - Insn.java (262 lines)
  - PrecedenceGraph.java (291 lines)
  - Template.java (347 lines)
  - Bundle.java (216 lines)
  - Schedule.java (946 lines)
  - Utils.java (88 lines)
  - Makefile (37 lines)
- Modules: The Basic Block Instruction Packing program consists of the following ten major modules, which are described further in the remainder of this section.
  - preprocessor (insntrans.awk)
  - configuration options (in Options.java)
  - architecture data (in resource.txt, template.txt)
  - main (in insnpack.java)
  - parser (in insnpack.java, Resource.java, Insn.java, Template.java)
  - datatypes (in Resource.java, Insn.java, Template.java, Bundle.java)
  - precedence graph (in PrecedenceGraph.java)
  - search (in insnpack.java)
  - schedule (in Schedule.java)
  - utility code (in Utils.java and JGL [13] library code)
- Preprocessor Module:[2] The insntrans.awk module preprocesses the IA-64 assembly language output of the GCC compiler, and outputs it in a format the insnpack Java program can read in as a list of instructions. The preprocessor is written in AWK, which is an excellent language for processing text using pattern matching. The following blocks form the AWK preprocessor.

[2]Thanks to Bart Massey for writing the original version of this module.

– block BEGIN: This block initializes an associative array which maps instruction names to instruction types [9] (e.g. $p[“addl”] = “a”).

– block to skip stops: If a stop is encountered at the beginning of the line, we skip this record.

– block to detect labels (which separate basic blocks): If a label (anything whose first field ends in a colon) is encountered, print “BB” to denote the beginning of a basic block, and go to the next record.

– blocks to skip assembler directives: If the first field of this record begins with a dot, then skip this record. Other assembler directives do not begin with a dot (e.g. stringz, data8, etc.). These also must be skipped.

– block to parse the operation: This block parses an instruction and prints formatted output in a form suitable for the insnpack.java module to read it in. An IA-64 assembly instruction begins with an optional predicate register in parentheses, e.g. ‘(p6)’, followed by a required operator name with optional extensions, followed by an optional list of read operands, followed by an equals sign, followed by an optional list of write operands, followed by an optional stop ‘;;’. An operand can be a register enclosed in a pair of square brackets ‘[ ]’, which implicitly indicates two operands: a read from the register and a read or write to memory, depending on which side of the equals sign the operand occurs. A predicate register in parentheses indicates the given predicate register is a read operand. One particular move instruction can read all predicate registers at once. The insnpack routine takes as input for each instruction a single character representing the instruction type, followed by a list of reads (implicit or explicit—this preprocessor makes them all explicit) separated by spaces, followed by an equals sign ‘=’, followed by a list of writes, followed by the entire text of the instruction enclosed in double quotes “ ”.

```
ALGORITHM: parse operation
BEGIN parse operation
   initialize counters and flags
   get the first field (predicate or operator)
   parse predicate and add to list of reads
   parse operator name and strip extensions
   parse the ‘mov’ operator (needs special type
    handling for different pseudo-ops)
   ensure instruction’s type is valid
   if no equals sign in insn writing = false
   for each operand
      if ‘=’ set writing = 0 and continue
      if ‘;;’ (stop), flag new insn group
      ignore labels which begin with a dot ‘.’
```

```
        strip trailing commas from operands
        if operand begins with square bracket,
         there are two operands: an explicit
         register read, and a read or write to
         memory, depending on whether we've seen
         a '=' yet
        ignore constants, globals, and macros
        translate register aliases
        ensure that operand is a valid register
        get reg name and store for printing later
         (if writing, in the writing list,
          else in the reading list)
      end for each operand
      print instruction's type
      print list of write operands, 'm' for memory
      print '='
      print list of read operands, 'm' for memory
      print separator character ';'
      print text of insn in double-quotes
    END parse operation
```

Because the GCC compiler is actually placing stops (thereby forming its own instruction groups) in its instruction stream, this module makes an assumption regarding the output of the GCC compiler, namely that there are no write-after-read conditions between instructions within instruction groups in the GCC assembly output. We verified that this assumption holds using the following debug code. We used functions to set and clear read-flags and write-flags for each resource. We cleared these flags at the beginning of each instruction group (i.e. at the beginning of the program and every time we encountered a stop). If a read of a resource occurred after a write of the same resource, then we detected a RAW condition. None of these conditions were detected.

- Options Module: The Options interface contains a set of option variables which determine the behavior of various aspects of the search engine.
  - MAX_NODES: node limit for search. Recommended setting: 1 million or less.
  - LIMITED_DISCREPANCY_SEARCH: enable / disable LDS (if disabled, depth-first search is used). Recommended setting: enabled.
  - LDS_MAX_LEVEL: the maximum level of LDS to try before switching to DFS. Recommended setting: 4 to 10. LDS expands the total number of nodes searched. A lower number for this field will keep the search space smaller.
  - PRUNE_BRANCH_AND_BOUND: enable / disable pruning provably suboptimal search subtrees. Recommended setting: true.

- STOP_ON_OPTIMAL: enable / disable quitting after finding a provably optimal solution. Recommended setting: true.
- PRINT_PARTIAL_ASSIGNMENTS: enable / disable printing improvements that are not complete schedules. If enabled, an improvement is defined as a schedule that is either more complete or of equal completeness with a better penalty cost. Recommended setting: false, in order to limit the amount of information displayed.
- PRINT_DOTS: enable / disable printing an "I'm alive" signal. Recommended setting: true.
- PRINT_GDB_STYLE: enable / disable printing assembly code in GDB's assembly style. If disabled, assembly code is printed Intel style. Recommended setting: true (GDB-style is more compact).
- DEBUG: enable / disable debug printing and checking legality of schedule. Recommended setting: true during development, false otherwise.
- VERBOSE: enable / disable verbose debug printing. Recommended setting: false.

- Architecture Data: (in resource.txt, template.txt). The first line in resource.txt contains the number of resources (e.g. 329). Each resource in resource.txt has two items: a name (r0-127, f0-127, p0-63, b0-7, m, ar.pfs) and a type (const, memory, branch, or normal). *Const* type means the register does not change value, and therefore multiple instructions in the same group can read from or write to it. This is the least restrictive resource type, because instructions with no other dependencies can be reordered. *Memory* means that any previous instruction in the same group which accesses this location affects this instruction. This type is more restrictive than const, because instructions with this kind of dependency cannot be reordered, but they can be in the same instruction group. *Branch* means that the register is normal. This type is just as restrictive as normal, and is present for historical reasons. *Normal* means that the register can either be read by multiple instructions in the same group, or written by at most one instruction in the group, but not both (except that it can be written by any instruction and then read by a branch). This is the most restrictive resource type.

  The file template.txt is organized as follows. The first line contains a single integer, listing the number of templates. The second line contains a single integer, listing the size of the templates, not including stops (i.e. '3' for IA-64). The remaining lines contain the templates of one of the following forms: ttt, tttt, or ttttt, where 't' is an element from {m,i,x,l,s}, and 's' is a stop.

- Main Module: (in insnpack.java) The main module interacts with the parser module and passes the results main module to the search module. There are three input files expected by the *insnpack* program: a resource

file, an insn file, and a template file. The resource file and template file are the same every time for the given architecture as described above, but the insn file differs for each basic block, as explained in the description of the preprocessor module. The main routine invokes the parser routine to read these files into data structures, then it initializes the cost threshold to the maximum integer value, creates an empty schedule, calculates a lower bound on the cost which may help it detect an optimal solution, and launches the search. If the LIMITED_DISCREPANCY_SEARCH option is enabled, then it loops repeatedly until reaching the maximum LDS cutoff value, and then it falls back into a depth-first search.

```
ALGORITHM launch search:
BEGIN launch search
   readFiles into Resource, Insn, and Template
    data structures and calculate PrecedenceGraph
    from Insns (parser module)
   initialize cost threshold to max integer value
   create empty Schedule (Schedule module)
   initialize cost lower bound (Schedule module)
   (calls to search module)
   if LIMITED_DISCREPANCY_SEARCH option
      possible = 2 * numInsns
      for cutoff=1 to min(possible,LDS-max-cutoff)
         search with cutoff, pass in empty-schedule
         reset schedule for next LDS iteration
      next cutoff
      if possible > LDS-max-cutoff
         search empty schedule with cutoff = possible
      endif
   else
      search empty schedule with cutoff = MAXINT
   endif
END launch search
```

- parser (in insnpack.java, Resource.java, Insn.java, Template.java) The parser module accepts an array of filenames and parses the files into appropriate Resource, Insn, Template and PrecedenceGraph data structures. The following methods form the parser module.

```
ALGORITHM readFiles (in insnpack.java)
BEGIN readFiles
   get a StreamTokenizer from resource
    filename (call Utils module)
   read #of resources from the file
    (call to Utils module)
   allocate an array of Resources
```

```
   for i=0 to number-of-resources - 1
      parse a resource
      store it in ith position of array
   next i
   get a StreamTokenizer from insn
    filename (call Utils module)
   read number-of-insns from the file
   allocate an array of Insns
   for i=0 to number-of-insns - 1
      parse an insn
      store it in ith position of array
      verify type not Insn.B if not last insn
   next i
   create precedence graph from insns and resources
    (call PrecedenceGraph module)
   get a StreamTokenizer from template
    filename (call Utils module)
   read #of templates from the file
   allocate an array of Templates
   for i=0 to number-of-templates - 1
      parse a template
      store it in ith position of array
   next i
   return arrays to main module
END readFiles
```

- parseResource (in Resource.java). This static method parses a resource from a single line in a text file, given a previously initialized StreamTokenizer and an integer identifier, and returns a newly constructed Resource object. The line in the resource file being parsed contains a resource name followed by a resource type. This method parses the line and invokes the Resource constructor to create a new Resource.

```
ALGORITHM parseInsn (in Insn.java)
BEGIN parseInsn
   state = 0
   while true
      get next token from stream tokenizer
      if EOL or EOF
         break
      if token is a single character
         make a string out of it
      else
         trim the whitespace from it
      switch state (no fallthrough)
      case state=0
```

```
            token holds insn type, store insn type
            state++
         case state=1:
            if '='
               state++
            else if ';'
               state+=2
            else
               compare token with all resource names
               if match, then set write flag for that resource
         case state=2:
            if ';'
               state++
            else
               compare token with all resource names
               if match, then set write flag for that resource
         case state=3:
            token holds entire insn text, store text
         end switch
      end while
      construct a new Insn from this information and return it
   END parseInsn

   ALGORITHM parseTemplate (in Template.java).
   BEGIN parseTemplate
      read a token from the stream tokenizer
      create an array of booleans representing stops
      init its elements to false
      p = 0
      for i=0 to tokenlength-1
         tokenchar = ith char of token
         if tokenchar == 's'
            pth element of stops array = true
         else
            set type according to tokenchar
      next i
      construct a new Template and return it
   END parseTemplate
```

- Datatypes (in Resource.java, Insn.java, Template.java, Bundle.java)
  - Resource (in Resource.java): For each resource, the Resource class encapsulates a name, identifier, and type of resource. Use of the same resource by multiple instructions can create dependencies between the instructions. The kind of dependency is sometimes determined by the type of resource. Legal values of the resource type are

Resource.C for constant (which means that writes to this resource do not change it), Resource.B for branch register, Resource.M for memory, and Resource.R for normal register (includes IA-64 general registers, floating-point registers, predicate registers, and application registers).

- Insn: The Instruction class maintains for each instance an instruction type, a set of resources read, a set of resources written, the actual instruction string, and an identifier representing the relative position in the original basic block. The relative position is important because it can be used to determine whether there is a precedence between two instructions. The precedence graph is a DAG.
- Template: Every instance of the Template class contains a template of slot types and stop positions. The member fields are an integer identifer, a template size, an array of template slot types with an associated array of type names, an array of stops, and the text of the template. The class provides a method implementing the match relation from instruction types to template types. There is an optional method for computing a heuristic score of this template with three instruction types, which is implemented in insnpack3.
- Bundle: Every instance of the Bundle class has a Template, a size which is the same as the size of the template, an array of Insns which are packed in this bundle (their types must match the template's slot types), an array of booleans indicating which positions in the bundle are occupied, and an array of booleans indicating which stops of the template are enabled. The reason that there is an array of booleans for the stops in the Bundle class as well as in the Template class is that for every template for the IA-64 that has a stop in a given position there is another identical template that does not have the stop in the given position. We made the implementation decision for insnpack3 to change from the template with a stop to a template without the stop simply by changing this flag, and were able to prune the search space from 24 templates to 10 (not 12, because of the two templates with stops in the middle).

  The Bundle class contains the following methods.

  * The Bundle constructor allocates the arrays, initializes the Insn array to null, the stops array to false, and the occupied array to false.
  * There is also a Bundle copy constructor which makes a copy that is exactly deep enough for our purposes. This copy constructor is used by the Schedule copy constructor, which is used every time an improved schedule is found. The Template and the primitive data types will never change so the template pointer is copied and the primitive data elements are copied, but the array of instructions must be newly allocated and each instruction copied

from the old bundle to the newly constructed bundle. Likewise the boolean arrays must be newly allocated and copied, because these elements change during the search. A deep copy is needed to preserve the copy when the original changes.

  - There are two methods called *addInsn*() and *removeInsn*(). The former adds an instruction to this bundle at the given position if the instruction matches the template at that position and the position is not occupied. The latter removes the instruction from the given position if the position is occupied.
  - There are two methods called *addStop*() and *removeStop*(). The former adds a stop to this bundle at the given position if the template has a stop at that position. The latter removes a stop from this bundle at the given position.
  - There is a method *stops*() that returns the number of stops in this bundle.
  - There is a method *nops*() that returns the number of nops in this bundle.
  - There is a method *nop*() that returns true if the given position in this bundle is not occupied. If the occupied flag for this slot in the bundle is false, the position might still be occupied if the template type of the previous slot in this bundle is of type L and that slot is occupied, so this case must also be checked.
  - There is a method *print*() that prints this bundle. Depending on how the Options are set, it will print this bundle GDB-style or Intel IA-64 assembly style.

- PrecedenceGraph Module (in PrecedenceGraph.java). The PrecedenceGraph class is a singleton for the insnpack program. Precedence information is computed once at the beginning of the search. The PrecedenceGraph instance contains the following data members:

  - There is an 2D integer array which stores precedence types between every pair of instructions as an adjacency matrix.
  - There is an array of "parent lists" for each instruction which is a adjacency list from each Insn to all its parents for fast lookup. The adjacency matrix is only used because there is an easy algorithm for computing transitive closure on this form of graph.

  The PrecedenceGraph class contains the following methods.

  - The precedence graph constructor calls functions that do all of the work. Given an array or instructions and an array of resources, it computes all of the precedences and builds the entire graph. Later uses of this module involve retrieving the parent list for a given instruction. The constructor allocates memory for the arrays and sets all precedences to C (none). Then it calls the *initialize*() method which calculates all the precedences.

– The *initialize*() method's algorithm is given here:

```
ALGORITHM initialize
BEGIN initialize
   for i=0 to numInsns - 1
      for j=i+1 to numInsns - 1
         type = computePrecedenceType(insn[i],insn[j])
         precedences[insn[i].id][insn[j].id] = type
      next j
   next i
   compute transitive closure
   construct lists
END initialize
```

– *computePrecedenceType*: There are four types of precedences: none (C=0), write-after-read (WAR=1), memory / branch (M=2), and register (R=3). The default precedence type between two instructions is initialized to C. If the first instruction has a higher or equal ID to the second instruction, then there is no precedence (C) in that direction. If the second instruction is a type B instruction, then the precedence type will be type memory / branch (M), but it may be upgraded later by the transitive closure. Each resource flag is examined in both instructions. If the second insn reads the same resource that the first insn writes (RAW), and the resource type is greater than the current type then the precedence type is upgraded to the resource type. If the second insn writes the same resource that the first insn writes (WAW) and the resource type is greater than the current precedence type, then the precedence type is upgraded to the resource type. If the second insn writes the same resource that the first insn reads (WAR) and the resource type is M, then the type is the larger of the given type and M. If the second insn writes the same resource that the first insn reads (WAR) and the resource type is not M, then the type is the larger of the given type and WAR.

– Here is the Floyd-Warshall algorithm for computing transitive closure [1] modified to compute transitive closure with multiple precedence types (larger types dominating):

```
ALGORITHM compute transitive closure
BEGIN compute transitive closure
   for k=0 to numInsns-1
      for i=0 to numInsns-1
         if precedences[i][k] > C
            for j=0 to numInsns - 1
               ij = precedences[i][j];
               ik = precedences[i][k];
               kj = precedences[k][j];
               if kj > C
```

```
                    precedences[i][j] =
                     max(ij,max(ik,kj))
                endif
             next j
          endif
       next i
    next k
 END compute transitive closure
```

  - There is a method *parentList*() to return a parent-list, given an instruction.

- Search module (in insnpack.java). The search code (in insnpack.java) depends heavily on the Schedule class (in Schedule.java). We first describe the *search*() function, then we describe the building blocks from the Schedule class which it uses.

  The *search*() function implements both a limited-discrepancy search [2] and a depth-first search; the search used depends on the Options module. This function can also prune provably suboptimal schedules and detect provably optimal schedules. The search is *layered*. This means that there are two kinds of choices which must be made in the search tree: template choices and instruction choices. The layering mechanism differs from insnpack2 to insnpack3.

```
ALGORITHM search (insnpack3)
INPUTS: schedule,group,depth,time,numDiscrep,cutoff
OUTPUTS: none, all schedules are printed to stdout
Note: '(Schedule)' denotes call to method in Schedule
Note: all lists are sorted by heuristic; the first
      element in list is the heuristic choice.
BEGIN search
   increment nodecount and quit if max exceeded
   if using LDS and numDiscrep > cutoff then return
   if using branch and bound pruning
      if cost (Schedule) > cost threshold then return
   if debugging then verify schedule is legal (Schedule)
   if schedule is improvement over best so far
      print schedule and copy it to best so far
      if schedule cost == cost lower bound then quit
   if schedule is complete then return (Schedule)
   // begin difference between insnpack2 & insnpack3
   if group is null or empty (JGL)
      group = legal insn group (Schedule)
   if schedule needs new bundle at given time (Schedule)
      get legal templates, given insn group (Schedule)
      for each template in legal templates (JGL)
```

```
      ndcopy = numDiscrep
      if template is not heuristic choice
         ndcopy = numDiscrep+1
      add a bundle to schedule
       w/given template (Schedule)
      get list of insns from group which
       match given position in template (Schedule)
      if list is empty (JGL)
         search schedule,group,depth,
          time+1,ndcopy,cutoff
      else
         for each insn in the list (JGL)
            ndcopy = numDiscrep
            if insn is not heuristic choice
               ndcopy = ndcopy+1
            if template is not heuristic choice
               ndcopy = ndcopy+1
            extend schedule by this insn (Schedule)
            groupCopy = group w/o insn
            if groupCopy is empty (JGL)
               place a stop after this insn (Schedule)
            search schedule,groupCopy,depth,
                   time+1,ndcopy,cutoff
            if placed a stop after this insn
               remove stop from stop time
            unextend schedule by this insn (Schedule)
         next insn
      endif
   next template
else // does not need new bundle
   get a list of insns from group which match
    position (given by time) in current template
   if list is empty
      search schedule,group,depth,
             time+1,numDiscrep,cutoff
   else
      for each insn in the list
         ndcopy = numDiscrep
         if insn is not heuristic choice
            ndcopy = numDiscrep+1
         extend schedule by this insn (Schedule)
         groupCopy = group w/o insn
         if groupCopy is empty
            place a stop after this insn (Schedule)
         search schedule,groupCopy,depth,
                time+1,ndcopy,cutoff
```

```
            if placed a stop after this insn
               remove stop from stop time
            unextend schedule by this insn (Schedule)
         next insn
      endif // insn list is/is not empty
   endif // needs/does not need new bundle
END search
```

- Schedule Module (in Schedule.java). The Schedule module maintains all the data and methods necessary to perform the following tasks:
  - create an empty partial schedule (constructor)
  - create a copy of an existing schedule; must deep-copy bundle list and scheduled instruction times
  - signify whether a schedule is complete (depth == numInsns)
  - signify whether a schedule needs a new bundle at the end before an instruction can be added at a given time (for insnpack3; time is an even multiple of bundle size)
  - signify whether an insn needs a new bundle at the end of the schedule before this insn can be added (for insnpack2)
  - signify whether a schedule is legal (for internal debugging)
  - get the template currently in use at a given time in the schedule
  - compute the earliest legal precedence time an insn can be added to the given schedule without adding any more bundles at the end of the schedule, assuming that all of this insn's predecessors have been scheduled (does not check for stops).
  - get a list of legal instructions (for insnpack2). This is a list of all instructions whose parents have all been scheduled. This list is sorted by a heuristic which rewards instructions which have early precedence times (see above).
  - get an instruction group (for insnpack3). This is a list of all instructions whose parents have all been scheduled, as well as all instructions whose parents have either already been scheduled or can be scheduled and the precedence type is M or WAR. This list is sorted by a heuristic which rewards instructions which have early precedence times (see above). Load or store instructions are put earlier in the list than non-memory instructions which have the same precedence time.
  - get a list of instructions from a given instruction group which match a given template at a given slot (for insnpack3).
  - get a list of legal templates which match a given group of instructions. This list is sorted by a template-selection heuristic which is scored as follows (for insnpack3 only): If the instruction group has

more than three instructions, count the number of instructions in the group which match this template in some slot. This is the template's heuristic score. However, if the instruction group has three or fewer instructions, then give 3 points for a match in the first slot, 2 points for a match in the second slot, and 1 point for a match in the third slot, making sure the matches are all in distinct slots.[3] Multiply this score by 3, and then if all instructions in the group can fit in distinct slots, add 2 bonus points for a template which can place a stop immediately after the last match and 1 bonus point for a template which can place a stop one slot after the last match.[4]

  - calculate and/or get the cost (6*stops+nops) of the current schedule
  - calculate the cost lower bound (6*stops) for optimality detection (calculated at construction time by counting the minimum number of instruction groups)
  - extend the current schedule by adding an instruction to it at a given time
  - unextend the current schedule by removing an instruction from it at a given time
  - extend the current schedule by adding an empty bundle with a given template at the end of the schedule
  - unextend the current schedule by removing an empty bundle from the end of the schedule
  - compute the first legal place an instruction can go in the given schedule without adding any more bundles at the end (for insnpack2).
  - add stop at a given time (position) in the schedule (for insnpack3; sets the bundle's stop flag at this position to true)
  - remove stop from a given time (position) in the schedule (for insnpack3)
  - print the current schedule

- Utility Code Module (in Utils.java and JGL [13] library code). The utility code in Utils.java contains code to compute and return the larger or smaller of two integers (min and max), to read an integer from a stream tokenizer which has already been initialized from an open file, and to initialize a stream tokenizer from a filename.

  The utility code from the JGL class libraries includes code to create and maintain a singly or doubly-linked list, to add elements to and delete elements from the list, to iterate through the list, and to get an element

[3] The template scores for groups of 3 or fewer instructions can be computed for each possible combination of three instruction types with the given template when the program is initialized.

[4] This heuristic idea is based on a simpler one from the Cygnus [18] GNU assembler, which does not involve reordering instructions and does not take stops into consideration. It simply scores templates by the number of instructions it can match in order.

from a given position in the list. The lists are used in returning lists of parent instructions for a given instruction from the precedence graph, maintaining lists of bundles which make up the parallelized instruction schedule, and returning lists of choices at each level of the search. The JGL library also provides a sort algorithm which could be applied to the lists, as long as the data elements of the list are instances of a class which implements a interface called *BinaryPredicate* to two instances of the class. This interface allows JGL to compare two objects in the list to determine which object should go first. The JGL sorting algorithm proved very useful for easy implementation of a value-ordering heuristic: We simply implemented a comparison function which compared two choices and gave a "score" to each choice, and the JGL sorting routine took care of the rest. The JGL library packages are well-documented using Javadoc. Using this code significantly increased development productivity.

4.3 Experimental Results

In this section, we compare experimental results of the different implementations of instruction packing with results obtained from GCC. Most basic blocks are small, and thus quite easily handled by our insnpack program. To find examples of long basic blocks which would be difficult even for state-of-the-art search techniques to tackle, we obtained a LINPACK benchmark program written in C. LINPACK has a lot of straight-line code, and we were able to find several lengthy basic blocks. Of course we found longer basic blocks in the unoptimized code than in the optimized code. We tested using both.

To perform the tests, we compiled linpack.c with the GCC compiler to obtain IA-64 assembly output. We chose six basic blocks of medium to long size. We then processed them with the preprocessor and then ran both versions of insnpack on those basic blocks. To get the scores for GCC, we ran the GDB debugger, disassembled the code, located the appropriate basic block, and counted stops and nops. Table 4.1 below compares the results between GCC, insnpack2, and insnpack3.[5] This table summarizes the data in Appendix A. Scores are $6 * stops + nops$. GCC code except insn0.txt was optimized with -O6. These results were obtained with the LIMITED_DISCREPANCY_SEARCH option turned off for insnpack2 and on for insnpack3.

There are several things we can learn from these results. The first thing to notice is that insnpack2 meets or beats GCC results in all but the largest basic block, and insnpack3 meats or beats insnpack2 and GCC results in every basic block, both in score and nodes searched (in fact, insnpack3 produces optimal results for every basic block). We believe these results can be explained by the

[5] Although the compiler is allowed to generate instruction schedules with bundle templates, GCC does not actually generate the bundled code. The bundles and therefore the scores should actually be attributed to the GNU assembler. Note that although a C compiler can defer this work to the assembler, a JIT compiler typically cannot because the machine code is created directly, without an assembly phase.

file	# insns	gcc score	insnpack2 score/nodes		insnpack3 score/nodes	
insn5.txt	10	14	14	62	+*12	11
insn4.txt	13	32	32	200K	*31	14
insn3.txt	14	61	58	903K	*57	17
insn2.txt	15	27	*18	1569	+*18	16
insn1.txt	15	48	48	27K	*44	18
insn0.txt†	26	115	121	91000K	*108	909

*believed optimal for instruction packing
+detected optimal solution
†from unoptimized GCC

Table 4.1: Scores comparing insnpack to optimized GCC.

differences in the search techniques used to arrive at the schedules. GCC uses a simple heuristic with no searching, insnpack2 uses a large search space with a little pruning, and insnpack3 uses a smaller search space with a lot of pruning, and with better template-selection heuristics.

For the largest block, we believe GCC beats insnpack2 because its template-selection heuristic works better than the template-selection heuristic in insnpack2. It is this belief that motivated the implementation of insnpack3 with a different search space. Also we believe that because insnpack2 allows template selection with more than the necessary number of stops, it spends a lot of time searching a part of the space with too many stops. Since we were using DFS, we found that the first template-selection decision in the search would never be reversed for this basic block.

We were quite pleased that insnpack3 beats GCC for every basic block we attempted, computing optimal results. We attribute the success of insnpack3 to template selection heuristic and to the use of LDS. The optimal result of 108 for insn0.txt is found by insnpack3 in LDS level 1, but its best result in DFS is 110. Optimal results are found very quickly in insnpack3 due to the excellent choice of template-selection heuristic. For all test cases except insn0.txt, insnpack3 finds an optimal schedule on its first try. Limited-discrepancy search causes insnpack3 to find optimal results on all our test cases in fewer than a thousand nodes.

One more thing is worthy of note. In insnpack3, we did not count nops that occured after the last non-nop instruction scheduled. This served to encourage schedules to move their nops to the end of the schedule, in addition to making it possible to detect a provably optimal solution in some cases. Scores that differ by one or two points in the table below are due to this fact. However for insn0.txt and insn1.txt, this is not much of a factor. We decided to leave this feature in the implementation of insnpack because the actual IA-64 architecture can have split issues between instructions in addition to those caused by stops.

In general, a schedule will be at least as fast if its nops are shifted to the

end as much as possible.[6] This is true for at least two reasons. First, if three or more nops can be shifted to the end of the schedule, the schedule can be shortened by eliminating bundles of three nops from the end. Second, a branch from an early slot in a bundle with a MBB or BBB template means that the later slots in the bundle will not be executed. If the nops follow a taken branch then they will not be executed, leading to a more efficient schedule.

4.4 Summary

We implemented two systematic search spaces for the insnpack problem, using simple DFS and LDS search with branch-and-bound pruning. We tested our implementations with medium-length basic blocks, which we expected to be more challenging for our programs to handle due to the large search space involved. Our first implementation, insnpack2, achieved better results than GCC in all but the largest of the basic blocks we attempted.

Spurred on by the challenge of improving the results on the largest basic block, we tackled a different search space. After improving our template selection heuristic and branch-and-bound pruning on insnpack3, we achieved rapid optimal results on all basic blocks we attempted.

[6] This will not always be the case. For example, if there is a dependency between a load from L1 cache into a register and a read from the same register, then the IA-64 architecture will delay for two cycles between the write and the read [11]. If a nop was placed so that the instruction which reads the register is already delayed for two cycles, then the IA-64 hardware will not need to insert extra delay into the pipeline.

Chapter 5

Trampolines

The last two chapters described instruction packing, which is an important component of an optimizing JIT compiler for IA-64. Another important component for the Kaffe JIT compiler for IA-64 is the trampoline subsystem, which is described in this chapter.

5.1 Introduction

Trampolines are springboards for just-in-time (JIT) compilation. Every time a method is invoked, the caller looks in a *dispatch table* for the address of the method's native code. If the method has not been translated yet, the dispatch table entry points to the method's trampoline function. When invoked, the trampoline function invokes the translator for the method and *fixes up* the dispatch table to point to the newly-loaded native code. This chapter explains how trampolines work and describes the requirements for implementing trampolines when porting Kaffe to IA-64. This chapter was extracted from [16], which also gives case studies of trampoline implementations for the SPARC and i386 architectures.

5.2 How Trampolines Work

When the JIT compiler begins compilation, it loads the first class and installs trampoline routines in the dispatch table for each method in the class (Figure 5.1).

The JIT compiler then translates the program's *main()* routine and transfers control to the program. When the program invokes a method, it looks up the method's address in its class's dispatch table. The first time a method is invoked, its dispatch table entry points to its trampoline, which was created when the class was loaded. For example, assume that method *bar2()* (in Figure 5.1) is the first method of its class to be called in a particular run of this program. This method's dispatch table entry points to the address of this method's trampoline.

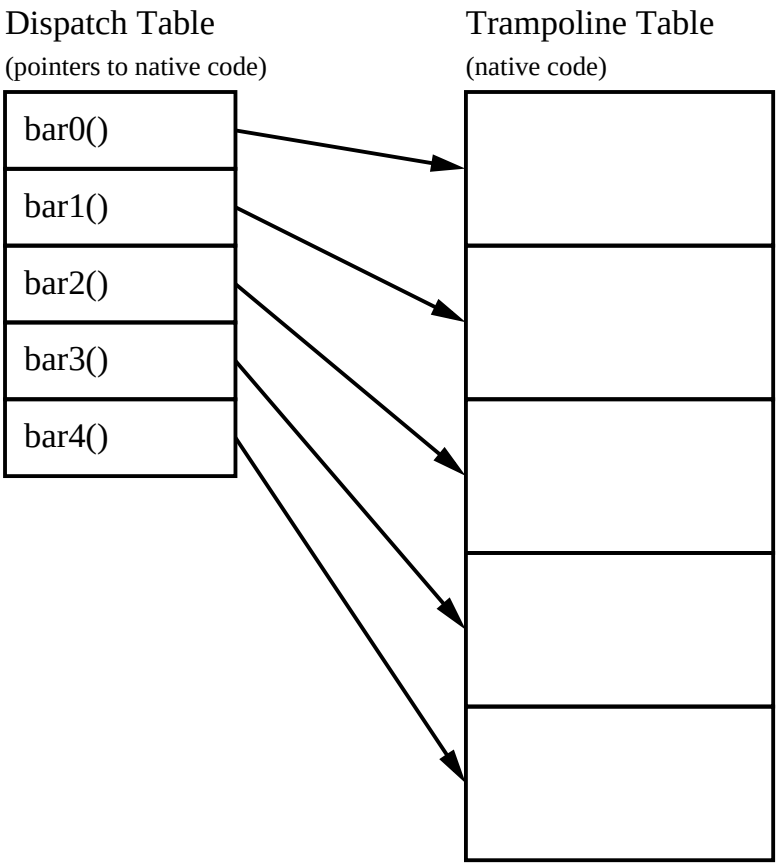

Figure 5.1: Tables at class load time.

The caller then calls the trampoline. (The trampoline itself contains enough information to know which method it needs to ask the translator to translate.) The trampoline invokes the translator, which translates the method and returns the code address of the resulting native code. The trampoline then fixes up the dispatch table to point to the method's native code (Figure 5.2).

Finally, the trampoline jumps to the newly-translated native code. When this method is called subsequently, the caller will follow the dispatch table entry to this method's native code.

Trampolines are transparent to both caller and callee. Because the caller does not realize that it is calling a trampoline, it cannot pass any special information, such as which method to translate, as an argument to the trampoline. Each method has its own trampoline so that this data can be stored in the trampoline itself. Because the callee does not know that it is being called by a trampoline, it must perform a normal return. The trampoline must ensure that the normal return passes control (and return values) back to the method's caller. Thus, there are six steps in the trampoline-based method-translation process:

1. At class load time, trampolines are created for each method and stored in the class's dispatch table. Every time a method is invoked, the caller looks up the method's address in its class's dispatch table.

2. The first time a method is invoked, the trampoline executes. The trampoline calls a fixup-trampoline method.

3. The fixup-trampoline method translates the callee method into native code and updates the environment so that control will return to the method's caller (not the trampoline) when the native-code method returns.

4. The fixup-trampoline method fixes up the dispatch table to point to the native code location.

5. The fixup-trampoline method *jumps* to the native-code.

6. The native code returns directly to the caller.

Figure 5.3 illustrates this process.

5.3 Trampoline Components And Porting Requirements

Most of the trampoline system is architecture dependent. Section 5.3.1 lists the principal components in the architecture-dependent part of the Kaffe trampoline system. Section 5.3.2 formalizes the requirements for trampoline implementation.

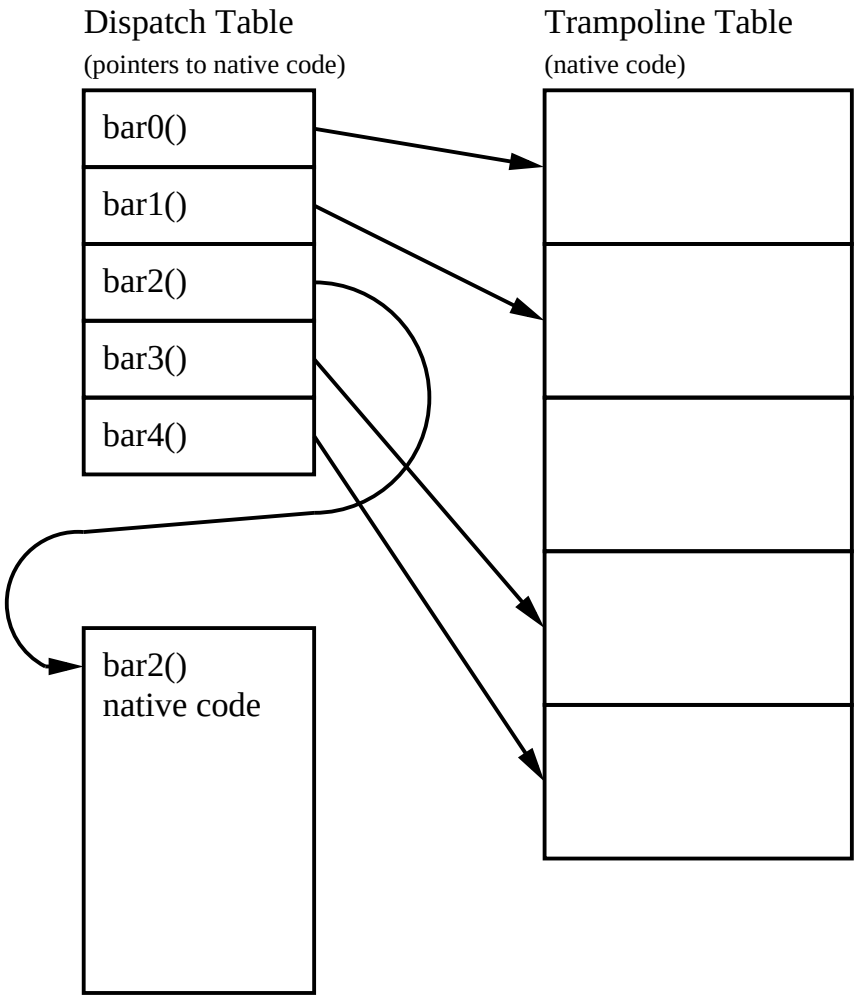

Figure 5.2: Tables after fix-up by trampoline.

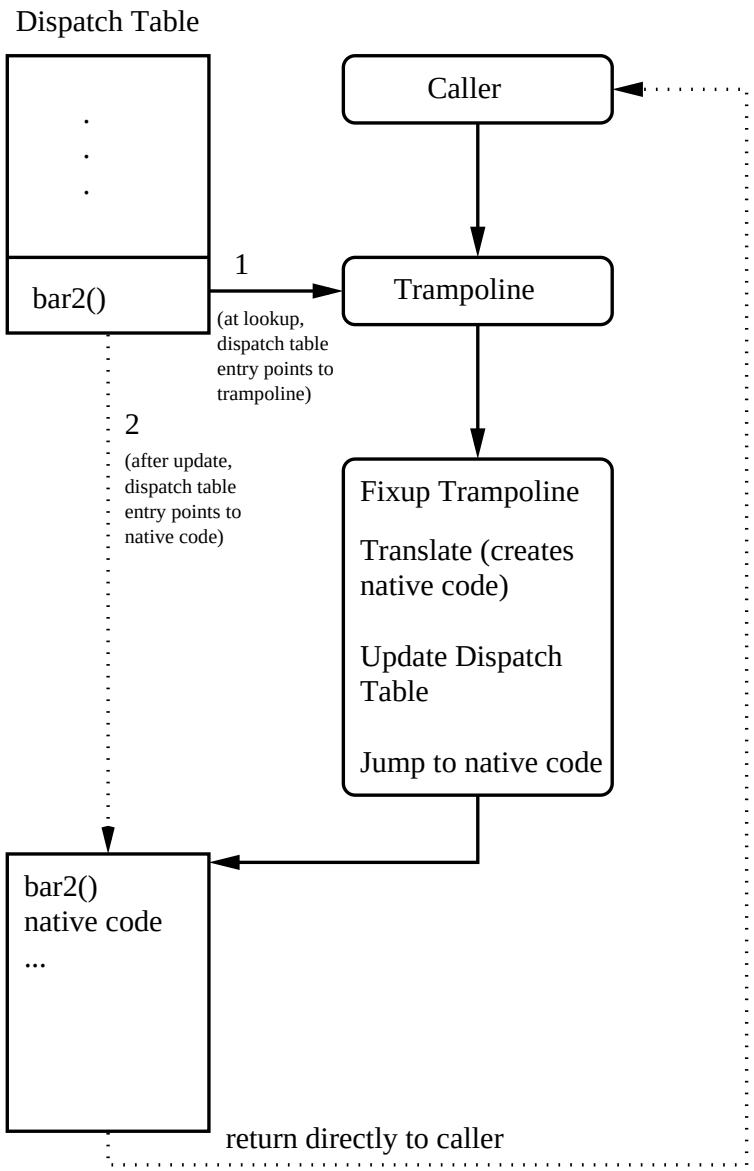

Figure 5.3: A call involving trampolines.

5.3.1 Trampoline Components

The developer porting Kaffe must implement the following trampoline-related architecture-dependent items:

- *struct _methodTrampoline*: trampoline code structure to be filled in
- *FILL_IN_TRAMPOLINE(t,m)* macro: fill in architecture-specific assembly code that will call *ia64_do_fixup_trampoline*
- *ia64_do_fixup_trampoline()* assembly macro: call *soft_fixup_trampoline()*, adjust the memory and register stack to make the trampoline transparent, and then *jump to* the resulting native code
- *FIXUP_TRAMPOLINE_DECL* macro: trampoline's method handle declaration: (*Method **_pmeth* (i386) or *Method *_meth (SPARC)*)
- *FIXUP_TRAMPOLINE_INIT* macro: trampoline's method handle definition: (*meth = *_pmeth* (i386) or *meth = _meth* (SPARC))
- *FLUSH_DCACHE(beg,end)* assembly macro: self-modifying code must flush data cache on some architectures.
- *sysdepCallMethod(callMethodInfo *call)*: put arguments on stack and / or registers as appropriate to the architecture, call the function (which may be a trampoline), store the return value (if any) in the *callMethodInfo* structure according to its type, and adjust the stack pointer after all done.

5.3.2 Porting Requirements

Here are the formal requirements for the trampoline infrastructure.

- Requirements for *struct _methodTrampoline*:
  1. The components of the trampoline structure must be packed to the alignment required by the architecture.
  2. The (optional) first main component of this structure is *code*, which is big enough to hold all necessary general machine instructions for a trampoline.
  3. The second main component of this structure is *fixup*, which is big enough to hold a call instruction (and a no-op instruction, if necessary).
  4. The third and final main component of this structure is *Method *meth*, which is a method descriptor that will be used by the translator to know which method to translate. This component must immediately follow the fixup component.
- Requirements for the trampoline in *FILL_IN_TRAMPOLINE(t,m)*:

1. The (optional) first component of a trampoline must be any code needed to communicate information to the *ia64_do_fixup_trampoline()* function. One piece of information the trampoline *must* communicate, either explicitly or implicitly, is the return address to the caller of the trampoline (e.g. The i386 architecture communicates the return address implicitly because the call instruction automatically places the return address on the stack; the SPARC architecture communicates this explicitly by copying the return address into a global variable.).
   FILL_IN_TRAMPOLINE(t,m) must contain code to fill in native code to do this, if needed.
2. The second component of a trampoline must be a relative call to the *ia64_do_fixup_trampoline()* function following the code to meet requirement 1. *FILL_IN_TRAMPOLINE(t,m)* must contain code to insert this *CALL* instruction into the trampoline's native code.
3. The last thing in a trampoline should be a *Method ** which points to the structure for the method to be translated. This is data that will never be executed, but can be accessed by *ia64_do_fixup_trampoline()* if requirement 1 is met.
   FILL_IN_TRAMPOLINE(t,m) must conclude with code to put this *Method ** in the trampoline.
4. A trampoline must not overwrite any callee-save registers or any callee-save data already on the memory or register stacks.
5. The fields filled in by this macro must exactly match those in *struct _methodTrampoline*

- Requirements for *FIXUP_TRAMPOLINE_DECL* and *FIXUP_TRAMPOLINE_INIT* macros:
  1. *FIXUP_TRAMPOLINE_DECL* must be either *Method *_meth* or *Method **_pmeth*.
  2. If *FIXUP_TRAMPOLINE_DECL* is *Method *_meth*, then *FIXUP_TRAMPOLINE_INIT* must be *meth = _meth)*; if *FIXUP_TRAMPOLINE_DECL* is *Method **_pmeth*, then *FIXUP_TRAMPOLINE_INIT* must be *meth = *_pmeth)*.
  3. The type of the method handle (*Method ** or *Method ***) in both of these macros must agree with the type passed from the trampoline to the *soft_fixup_trampoline()* routine. *Method ** should be used if the architecture stores return addresses in a register; *Method *** should be used if return addresses are stored on the stack.

- Requirements for *ia64_do_fixup_trampoline()*:
  1. *ia64_do_fixup_trampoline()* must retrieve the *Method ** or a pointer to it from the information obtained from the trampoline.

2. *ia64_do_fixup_trampoline()* must call *soft_fixup_trampoline()* and pass the *Method ** or pointer to it as the first and only argument. Note: *soft_fixup_trampoline()* will update the dispatch table for the newly-translated method.
3. *ia64_do_fixup_trampoline()* must retrieve the return value from *soft_fixup_trampoline()* and ensure that it is in a register that can be used to address the destination to a jump instruction.
4. *ia64_do_fixup_trampoline()* must restore the memory and/or register stack, the arguments to the newly translated native code method, and all other computer state information such that a call to the native code will return to the caller of the trampoline (and thus will never return to the trampoline or to *ia64_do_fixup_trampoline()*) This is the "cleanup" or "fixup" requirement.

- Requirements for *FLUSH_DCACHE(beg,end)*:
  1. *FLUSH_DCACHE()* must be implemented if data cache must be flushed for self-modifying code on the architecture.
  2. *FLUSH_DCACHE()* if implemented, must flush all data cache locations from *int *beg* to *int *end*
  3. *FLUSH_DCACHE()*, if not implemented, must be an empty macro.
- Requirements for *sysdepCallMethod()* C function containing inline assembly code:
  1. *sysdepCallMethod()* must have the following signature:
     *static inline void sysdepCallMethod(callMethodInfo *call)*
  2. *sysdepCallMethod()* must not contain any code specific to trampolines (i.e. It must assume the native code is already translated).
  3. *sysdepCallMethod()* must emit inline GCC assembly code to move arguments from its *callMethodInfo ** argument list into the appropriate registers or stack locations for a function call in the architecture. This may include pushing arguments onto the stack, allocating stack space, and/or saving register windows
     (i.e. *sysdepCallMethod()* must prepare for a call, according to the IA-64 calling conventions).
  4. *sysdepCallMethod()* must switch based on the return type for the function, call the function, and store the return value in the appropriate *jint*, *jfloat*, *jdouble*, or *jlong* member of *call-¿ret* if the return value is not *void*.
  5. *sysdepCallMethod()* must emit inline GCC assembly code to perform the architecture-specific operations which return from a call. This may include deallocating stack space and/or restoring register windows.

5.4 Summary

This chapter described how trampolines work, both on the abstract conceptual level and in the Kaffe implementation. It also listed porting requirements. Our full paper on trampolines [16] provides further information, giving two case studies of the Kaffe SPARC and i386 implementations.

Chapter 6

Future Work

Listed here are some worthwhile activities toward completing an optimizing JIT compiler for IA-64.

- Enhance insnpack2 and insnpack3 implementations. The insnpack2 implementation needs to be extended to handle WAR dependencies for completeness. It also needs better calculation of the cost function for partial schedules and better pruning of schedules that have too many stops. Template-selection heuristics need improvement as already implemented in insnpack3. These fixes may make insnpack2 competitive with insnpack3, while achieving completeness (which insnpack3 does not).

  The insnpack3 implementation would be improved by adding code for pruning instruction choices for instruction groups which do not have any M or WAR precedences between instructions. The template-selection heuristic should be improved to break ties between templates that accomodate the current instruction group with slots to spare after an internal stop. The ties should be broken by examining the next instruction group. This will be a little complicated, but it will allow the optimal solution for insn0.txt to be detected on the first try.

- Extend insnpack to calculate cycle times (split issues, etc.). While we believe our implementation provides a pretty good first approximation to the costs born by various basic block instruction packing schedules, the split issue conditions described in [11] would give a more accurate measure of performance.

- Experiment further using different scheduling and search techniques. We have only used systematic search techniques with surprisingly good results. However, for some large basic blocks, we still compute sub-optimal answers. Non-systematic *local search* techniques may work even better on these problems.

- Extend insnpack to be able to optimize whole methods. Instead of reading in a single basic block, insnpack should read in a whole method, which

has been conveniently broken into a sequence of basic blocks. Then it should optimize each basic block and return a sequence of schedules, one for each basic block. Since the output is machine code, it will have to interface correctly with the icode routines for patching labels into machine addresses.

- Extend insnpack to be able to handle rotating registers and all instructions involving implicit application registers. Ensure that floating-point exceptions are handled properly even with code reordering.
- Integrate insnpack into the JIT compiler (Obviously this must be done after implementing the Kaffe icode and trampolines). Some sort of instruction packing must be implemented for a JIT to run on IA-64.
- Implement Kaffe trampolines for IA-64 as described here and in [16]. The trampoline method translation subsystem must be implemented in IA-64 assembly and / or machine language.
- Implement Kaffe icode instructions for IA-64 as described in [15]. The Kaffe [12] JIT compiler is independent of the machine architecture up to the intermediate-code level, which is called "icode". Each required icode instruction must be implemented in IA-64 machine language.

We have made significant strides toward an optimizing JIT compiler for IA-64. There is a lot of interesting work left to do.

Chapter 7

Conclusions

This chapter summarizes the conclusions we have drawn from this project.

- The IA-64 architecture has different requirements than most current processors, leading to new kinds of search problems. The instruction packing problem requires layered search, which can make the search implementation more challenging.

- There is not always enough time for complete search, but we have shown that some search is better than none and that it is possible to do anytime search. Anytime search can easily be accomplished in our implementations of the IA-64 instruction packing problem by setting a limit on the number of nodes to search for each basic block. Our best implementation of insnpack found optimal results on all the test cases we tried in less than a second, including one test case with 26 instructions (long for a basic block). We consider this approach to be quite viable for JIT compilers, as well as for whole-program compilers such as GCC.

- Techniques which significantly improved search results in instruction packing are reformulating the search space, improving the cost estimate for partial schedules while using branch-and-bound pruning, improving the value-ordering (template-selection and instruction-selection) heuristics, and using LDS. Future improvements to the value-ordering heuristics and to instruction pruning are possible.

- Search techniques provide a mechanism for close examination and tuning of value-ordering heuristics, which can greatly improve results.[1] From insnpack2 to insnpack3 the template-selection heuristic was improved, and immediately insnpack3 finds optimal results on all test cases with all but one of these being found in LDS0. Moreover, the test case in which the

[1] This observation came from one of the attendees of the project presentation, who noted that optimal results were found by the heuristics alone in nearly all cases.

optimal result was not found in LDS0 was found in LDS1, and the error in the heuristics was in the instruction-selection heuristic, not in the template-selection heuristic.

- JIT compilation has different requirements than whole-program compilation, making a method-by-method compilation technique such as trampolines necessary.

Acknowledgements

I would like to thank the following people, without whom this project would not have been possible. Professor Bart Massey advised me on this project and helped me every step of the way (this is really an understatement); he has also taught me nearly everthing I know about search and scheduling. Professor Cynthia Brown reviewed this paper and made helpful suggestions at critical project decision points. Professor Bart Massey, Professor Cynthia Brown, Professor Andrew Tolmach, and Keith Packard of SuSE attended my project presentation. George Borden worked with me on a class project called MD4; I used some of our source code as a starting point for implementing insnpack. Friends from my church have prayed for me for years. My parents, Gary and Wanda Sanseri, taught me at home through high school and have wisely counseled me during my early adult years. I am especially grateful to my wife Marissa for faithfully encouraging me even when she felt like a 'computer science widow.' This project is dedicated to God who is worthy of all honor and glory and has given me the strength and ability to do this work. *Soli Deo gloria.*

Appendix A

Experimental Results

This appendix provides complete results for the test cases summarized in Table 4.1. For each basic block, a filename is given, followed by the GCC, insnpack2, and insnpack3 results, respectively. Stops are denoted by a pair of consecutive semicolons (;;). The instructions for GCC are not numbered; numbering each instruction in GCC's results, starting with 0 and skipping nops, gives the instruction numbers used in the insnpack2 and insnpack3 results. Nops following the last non-nop instruction are not counted in the score.

The insnpack2 implementation outperforms GCC for all cases except for insnpack0.txt which is the longest basic block. The insnpack3 implementation finds believed-optimal schedules for all basic blocks and outperforms GCC in all cases.

```
================
insn5.txt results:

GCC: 2 stops + 2 nops
[MMI]       mov r1=r41
            stfd [r40]=f8
            mov r56=1;;
[MMF]       mov r40=r1
            mov r53=r42
            fmerge.ns f8=f8,f8
[MII]       shladd r55=r39,3,r45
            mov r57=r44
            mov r58=r56
[MIB]       nop.m 0x0
            nop.i 0x0
            br.call.sptk.many b0=0x2b60 <daxpy_r>;;

insnpack2: 2 stops + 2 nops
cost: 14
```

```
depth: 10
nodes examined: 62
[mii]   0       mov r1 = r41
        2       addl r56 = 1, r0
        4       mov r53 = r42
[mii]   1       stfd [r40] = f8
        6       shladd r55 = r39, 3, r45
        7       mov r57 = r44
[misi]          nop.m
                nop.i
                ;;
        8       mov r58 = r56
[mfbs]  3       mov r40 = r1
        5       fneg f8 = f8
        9       br.call.sptk.many b0 = daxpy_r#
                ;;

insnpack3: 2 stops + 0 nops
cost: 12
depth: 10
discrepancies: 0
nodes examined: 11
This is an optimal schedule.
search stopped after finding optimal solution.
[mfi]   0       mov r1 = r41
        5       fneg f8 = f8
        2       addl r56 = 1, r0
[mmi]   1       stfd [r40] = f8
        4       mov r53 = r42
        6       shladd r55 = r39, 3, r45
[msmi]  7       mov r57 = r44
                ;;
        3       mov r40 = r1
        8       mov r58 = r56
[bbbs]  9       br.call.sptk.many b0 = daxpy_r#
                nop.b
                nop.b
                ;;

================
insn4.txt results:

GCC: 5 stops + 2 nops
[MII]       setf.sig f96=r42
            mov r55=1
            mov r53=r42;;
```

```
[MFI]       nop.m 0x0
            xmpy.l f96=f16,f96
            mov r41=r1
[MMI]       mov r56=r44;;
            getf.sig r39=f96
            mov r57=r55
[MMI]       shladd r40=r42,3,r44;;
            nop.m 0x0
            sxt4 r14=r39;;
[MIB]       shladd r54=r14,3,r45
            add r39=r42,r39
            br.call.sptk.many b0=0x2dd0 <ddot_r>;;

insnpack2: 5 stops + 2 nops
cost: 32
depth: 13
nodes examined: 199725
[mii]   0       setf.sig f96 = r42
        1       addl r55 = 1, r0
        2       mov r53 = r42
[misi]  4       mov r41 = r1
        5       mov r56 = r44
                ;;
        8       shladd r40 = r42, 3, r44
[mfis]  7       mov r57 = r55
        3       xma.l f96 = f16, f96, f0
                nop.i
                ;;
[misis] 6       getf.sig r39 = f96
                nop.i
                ;;
        9       sxt4 r14 = r39
                ;;
[mibs]  10      shladd r54 = r14, 3, r45
        11      add r39 = r42, r39
        12      br.call.sptk.many b0 = ddot_r#
                ;;

insnpack3: 5 stops + 1 nop
cost: 31
depth: 13
discrepancies: 0
nodes examined: 14
[mmi]   0       setf.sig f96 = r42
        1       addl r55 = 1, r0
        2       mov r53 = r42
```

```
[mmis]  4       mov r41 = r1
        5       mov r56 = r44
        8       shladd r40 = r42, 3, r44
                ;;
[mfis]  7       mov r57 = r55
        3       xma.l f96 = f16, f96, f0
                nop.i
                ;;
[msmis] 6       getf.sig r39 = f96
                ;;
        11      add r39 = r42, r39
        9       sxt4 r14 = r39
                ;;
[mbbs]  10      shladd r54 = r14, 3, r45
        12      br.call.sptk.many b0 = ddot_r#
                nop.b
                ;;

================
insn3.txt results:

GCC: 9 stops + 7 nops
[MII]       sub r14=r46,r43
            mov r41=r1;;
            sxt4 r42=r14;;
[MMF]       setf.sig f97=r42
            shladd r40=r42,3,r44
            nop.f 0x0;;
[MFI]       nop.m 0x0
            xmpy.l f97=f16,f97
            nop.i 0x0
[MMI]       ldfd f8=[r40];;
            getf.sig r39=f97
            nop.i 0x0;;
[MMI]       add r14=r42,r39;;
            nop.m 0x0
            sxt4 r14=r14
[MII]       nop.m 0x0
            sxt4 r39=r39;;
            shladd r14=r14,3,r45;;
[MIB]       ldfd f9=[r14]
            nop.i 0x0
            br.call.sptk.many b0=0x1990 <dgesl+592>;;

insnpack2: 9 stops + 4 nops
cost: 58
```

```
depth: 14
nodes examined: 902771
[misi]  0       sub r14 = r46, r43
        1       mov r41 = r1
                ;;
        2       sxt4 r42 = r14
[msmis]         nop.m
                ;;
        3       setf.sig f97 = r42
        4       shladd r40 = r42, 3, r44
                ;;
[mfis]  6       ldfd f8 = [r40]
        5       xma.l f97 = f16, f97, f0
                nop.i
                ;;
[misis] 7       getf.sig r39 = f97
                nop.i
                ;;
        8       add r14 = r42, r39
                ;;
[misis]         nop.m
        9       sxt4 r14 = r14
                ;;
        11      shladd r14 = r14, 3, r45
                ;;
[mibs]  12      ldfd f9 = [r14]
        10      sxt4 r39 = r39
        13      br.call.sptk.many b0 = __divdf3#
                ;;

insnpack3: 9 stops + 3 nops
cost: 57
depth: 14
discrepancies: 0
nodes examined: 17
[misis] 0       sub r14 = r46, r43
        1       mov r41 = r1
                ;;
        2       sxt4 r42 = r14
                ;;
[misi]  3       setf.sig f97 = r42
        4       shladd r40 = r42, 3, r44
                ;;
                nop.i
[mfis]  6       ldfd f8 = [r40]
        5       xma.l f97 = f16, f97, f0
```

```
                nop.i
                ;;
[msmis] 7       getf.sig r39 = f97
                ;;
        8       add r14 = r42, r39
        10      sxt4 r39 = r39
                ;;
[misis]         nop.m
        9       sxt4 r14 = r14
                ;;
        11      shladd r14 = r14, 3, r45
                ;;
[mbbs]  12      ldfd f9 = [r14]
        13      br.call.sptk.many b0 = __divdf3#
                nop.b
                ;;

================
insn2.txt results:

GCC: 4 stops + 3 nops
[MII]       alloc r52=ar.pfs,27,21,0
            mov r50=r12
            nop.i 0x0
[MII]       setf.sig f96=r33
            mov r51=b0;;
            adds r12=-32,r12
[MII]       mov r45=r32
            mov r49=r35;;
            adds r2=-16,r50
[MII]       mov r44=r36
            sxt4 r47=r34;;
            nop.i 0x0
[MII]       stf.spill [r2]=f16,16
            cmp4.eq p6,p7=0,r38
            sxt4 r14=r37
[MFB]       nop.m 0x0
            fsxt.r f16=f96,f96
      (p06) br.cond.dptk.few 0x1c10 <dgesl+1232>;;

insnpack2: 3 stops + 0 nops
cost: 18
depth: 15
nodes examined: 1569
[mii]   0       alloc r52 = ar.pfs, 7, 14, 6, 0
        1       mov r50 = r12
```

```
        3       mov r51 = b0
[mii]   2       setf.sig f96 = r33
        5       mov r45 = r32
        6       mov r49 = r35
[mii]   8       mov r44 = r36
        9       sxt4 r47 = r34
        11      cmp4.eq p6, p7 = 0, r38
[misis] 4       adds r12 = -32, r12
        12      sxt4 r14 = r37
                ;;
        7       adds r2 = -16, r50
                ;;
[mfbs]  10      stf.spill [r2] = f16, 16
        13      fsxt.r f16 = f96, f96
        14      (p6) br.cond.dptk .L120
                ;;

insnpack3: 3 stops + 0 nops
cost: 18
depth: 15
discrepancies: 0
nodes examined: 16
This is an optimal schedule.
search stopped after finding optimal solution.
[mmi]   0       alloc r52 = ar.pfs, 7, 14, 6, 0
        1       mov r50 = r12
        3       mov r51 = b0
[mmi]   2       setf.sig f96 = r33
        4       adds r12 = -32, r12
        5       mov r45 = r32
[mmi]   6       mov r49 = r35
        8       mov r44 = r36
        9       sxt4 r47 = r34
[misis] 11      cmp4.eq p6, p7 = 0, r38
        12      sxt4 r14 = r37
                ;;
        7       adds r2 = -16, r50
                ;;
[mfbs]  10      stf.spill [r2] = f16, 16
        13      fsxt.r f16 = f96, f96
        14      (p6) br.cond.dptk .L120
                ;;

================
insn1.txt results:
```

```
GCC: 7 stops + 6 nops
[MMI]       setf.sig f96=r42
            setf.sig f97=r42
            nop.i 0x0;;
[MFI]       adds r15=8,r15
            xma.l f97=f16,f97,f96
            mov r56=1
[MMI]       sub r53=r48,r42;;
            getf.sig r14=f97
            mov r39=r1
[MMI]       add r57=r44,r15;;
            nop.m 0x0
            sxt4 r14=r14
[MMI]       mov r58=r56;;
            nop.m 0x0
            dep.z r14=r14,3,61;;
[MMI]       adds r14=8,r14;;
            add r55=r45,r14
            nop.i 0x0
[MIB]       nop.m 0x0
            nop.i 0x0
            br.call.sptk.many b0=0x2b60 <daxpy_r>;;

insnpack2: 7 stops + 6 nops
cost: 48
depth: 15
nodes examined: 26589
[mii]   0       setf.sig f96 = r42
        2       adds r15 = 8, r15
        4       addl r56 = 1, r0
[mii]   1       setf.sig f97 = r42
        5       sub r53 = r48, r42
        7       mov r39 = r1
[misi]          nop.m
                nop.i
                ;;
        8       add r57 = r44, r15
[mfis]  10      mov r58 = r56
        3       xma.l f97 = f16, f97, f96
                nop.i
                ;;
[misis] 6       getf.sig r14 = f97
                nop.i
                ;;
        9       sxt4 r14 = r14
                ;;
```

```
[misis]         nop.m
        11      shl r14 = r14, 3
                ;;
        12      adds r14 = 8, r14
                ;;
[mibs]  13      add r55 = r45, r14
                nop.i
        14      br.call.sptk.many b0 = daxpy_r#
                ;;

insnpack3: 7 stops + 2 nops
cost: 44
depth: 15
discrepancies: 0
nodes examined: 18
[mmi]   0       setf.sig f96 = r42
        1       setf.sig f97 = r42
        2       adds r15 = 8, r15
[mmis]  4       addl r56 = 1, r0
        5       sub r53 = r48, r42
        7       mov r39 = r1
                ;;
[mfis]  8       add r57 = r44, r15
        3       xma.l f97 = f16, f97, f96
        10      mov r58 = r56
                ;;
[msmis] 6       getf.sig r14 = f97
                ;;
                nop.m
        9       sxt4 r14 = r14
                ;;
[misis]         nop.m
        11      shl r14 = r14, 3
                ;;
        12      adds r14 = 8, r14
                ;;
[mbbs]  13      add r55 = r45, r14
        14      br.call.sptk.many b0 = daxpy_r#
                nop.b
                ;;

================
insn0.txt results:

GCC: 18 stops + 7 nops
[MII]       mov r15=r33
```

```
            adds r14=-84,r33;;
            mov r15=r33
[MMI]       adds r16=-84,r33;;
            ld4 r15=[r16]
            nop.i 0x0;;
[MII]       nop.m 0x0
            extr r17=r15,31,1;;
            mov r16=r17;;
[MII]       nop.m 0x0
            extr.u r17=r16,31,1;;
            mov r16=r17;;
[MMI]       add r15=r15,r16;;
            nop.m 0x0
            extr r16=r15,1,31;;
[MMI]       mov r15=r16;;
            st4 [r14]=r15
            mov r15=r33
[MMI]       adds r14=-84,r33;;
            mov r15=r33
            adds r16=-84,r33;;
[MMI]       ld4 r17=[r16];;
            mov r15=r17
            nop.i 0x0;;
[MMI]       add r16=r15,r17;;
            st4 [r14]=r16
            mov r14=r33
[MMI]       adds r15=-84,r33;;
            ld4 r14=[r15]
            nop.i 0x0;;
[MIB]       cmp4.lt p6,p7=9,r14
            nop.i 0x0
      (p06) br.cond.dptk.few 0x300 <mempool+768>;;

insnpack2: 17 stops + 19 nops
cost: 121
depth: 26
nodes examined: 91020541
[mii]   0       mov r15 = r33
        1       adds r14 = -84, r33
        3       adds r16 = -84, r33
[misis]         nop.m
                nop.i
                ;;
        2       mov r15 = r33
                ;;
[misi]  4       ld4 r15 = [r16]
```

```
                nop.i
                ;;
        5       extr r17 = r15, 31, 1
[misi]          nop.m
                nop.i
                ;;
        6       mov r16 = r17
[misi]          nop.m
                nop.i
                ;;
        7       extr.u r17 = r16, 31, 1
[misi]          nop.m
                nop.i
                ;;
        8       mov r16 = r17
[misi]          nop.m
                nop.i
                ;;
        9       add r15 = r15, r16
[misi]          nop.m
                nop.i
                ;;
        10      extr r16 = r15, 1, 31
[misi]          nop.m
                nop.i
                ;;
        11      mov r15 = r16
[msmi]  16      adds r16 = -84, r33
                ;;
        12      st4 [r14] = r15
        13      mov r15 = r33
[msmis] 14      adds r14 = -84, r33
                ;;
        17      ld4 r17 = [r16]
        15      mov r15 = r33
                ;;
[misi]  18      mov r15 = r17
                nop.i
                ;;
        19      add r16 = r15, r17
[msmis] 22      adds r15 = -84, r33
                ;;
        20      st4 [r14] = r16
        21      mov r14 = r33
                ;;
[misi]  23      ld4 r14 = [r15]
```

```
                nop.i
                ;;
        24      cmp4.lt p6, p7 = 9, r14
[mibs]          nop.m
                nop.i
        25      (p6) br.cond.dptk .L11
                ;;

insnpack3: 17 stops + 6 nops
cost: 108
depth: 26
discrepancies: 1
nodes examined: 909
[mmis]  0       mov r15 = r33
        1       adds r14 = -84, r33
        3       adds r16 = -84, r33
                ;;
[msmis] 2       mov r15 = r33
                ;;
        4       ld4 r15 = [r16]
                nop.i
                ;;
[misis]         nop.m
        5       extr r17 = r15, 31, 1
                ;;
        6       mov r16 = r17
                ;;
[misis]         nop.m
        7       extr.u r17 = r16, 31, 1
                ;;
        8       mov r16 = r17
                ;;
[msmis] 9       add r15 = r15, r16
                ;;
                nop.m
        10      extr r16 = r15, 1, 31
                ;;
[misi]  11      mov r15 = r16
        16      adds r16 = -84, r33
                ;;
        13      mov r15 = r33
[mmis]  12      st4 [r14] = r15
        17      ld4 r17 = [r16]
        14      adds r14 = -84, r33
                ;;
[msmis] 15      mov r15 = r33
```

```
                ;;
        18      mov r15 = r17
                nop.i
                ;;
[misi]  19      add r16 = r15, r17
        22      adds r15 = -84, r33
                ;;
        21      mov r14 = r33
[msmis] 20      st4 [r14] = r16
                ;;
        23      ld4 r14 = [r15]
                nop.i
                ;;
[mbbs]  24      cmp4.lt p6, p7 = 9, r14
        25      (p6) br.cond.dptk .L11
                nop.b
                ;;
```

Bibliography

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 1990.

[2] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615, San Mateo, August 20–25 1995. Morgan Kaufmann. ISBN 1-55860-363-8.

[3] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *Lecture Notes in Computer Science*, 641:176, 1992. ISSN 0302-9743.

[4] Hewlett Packard, Inc. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. Web document, 1999. URL `http://www.trimaran.org`. This is the website for the open source distribution of the Trimaran infrastructure for research in instruction-level parallelism, specifically designed for researchers investigating Explicitly Parallel Instruction Computing (EPIC), high-performance computing systems, instruction-level parallelism, compiler optimizations, computer architecture, adaptive and embedded systems, and language design.

[5] Kiyo Inaba. What is Trampoline Code in Kaffe? Web document, 1998. URL `http://www2.biglobe.ne.jp/~inaba/trampolines.html`.

[6] Intel, Inc. *IA-64 Application Developer's Architecture Guide*, May 1999. Order Number: 245188-001. This was our main source of information for the instruction packing problem. It was replaced by the four-volume IA-64 Architecture Software Developer's Manual [9].

[7] Intel, Inc. *IA-64 Software Conventions and Runtime Architecture Guide*, August 1999. Order Number: 245256-001.

[8] Intel, Inc. *Intel Architecture Software Developer's Manual, Volumes 1-3: Basic Architecture, Instruction Set Reference, and System Programming Guide*, 1999. Order Numbers 243190, 243191, and 243192. This manual is for the IA-32 architecture.

[9] Intel, Inc. *IA-64 Architecture Software Developer's Manual, Volumes 1-4: Application Architecture, System Architecture, Instruction Set Reference, and Itanium Processor Programmer's Guide*, January 2000. Order Numbers: 245317-001, 245318-001, 245319-001, and 245320-001.

[10] Intel, Inc. *IA-64 Assembly Language Reference Guide*, January 2000. Order Number: 245363-001.

[11] Intel, Inc. *Itanium Processor Microarchitecture Reference for Software Optimization*, March 2000. Order Number: 245473-001. This will be most useful in a future version of our instruction packing program which makes calculations based on actual cycle types.

[12] Kaffe. A Cleanroom, Open Source Implementation of a Java Virtual Machine and Class Libraries. Web document, 2000. URL `http://www.kaffe.org`. This is the website for the open source distribution of the Kaffe Java Virtual Machine and class libraries, including Kaffe's Java JIT compiler. The latest release to date is version 1.0.5 (but a CVS tree is available).

[13] ObjectSpace, Inc. Java Generic Library. Web document, 2000. URL `http://www.objectspace.com/products/prodJGL.asp`. This is the website for downloading ObjectSpace's Java Generic Library called ObjectSpace JGL. ObjectSpace JGL is a powerful add-on for the JDK that provides a series of advanced collections and more than 50 generic algorithms. JGL is designed to complement, not replace, the basic features found in the JDK and is ideal for enterprise Java developers. JGL 3.1 enhanced the JGL offering with distributed collection support, allowing the remote construction, access, and persistence of all JGL containers using ObjectSpace Voyager.

[14] Richard P. Paul. *SPARC Architecture, Assembly Language Programming, & C.* Prentice Hall, 1994.

[15] Samuel Sanseri. The Kaffe JIT icode Instruction Set. URL `http://www.cs.pdx.edu/~sanseri/kaffe/k1.html`. This web page contains a table describing the Kaffe JIT instruction set., December 1999.

[16] Samuel Sanseri. Trampolines in the Kaffe JIT Compiler. URL `http://www.cs.pdx.edu/~sanseri/kaffe/k2.html`. This web page contains a description of Kaffe trampolines., February 2000.

[17] Silicon Graphics, Inc. SGI Pro64: A Suite of Optimizing Compiler Development Tools for Linux Intel Itanium Systems. Web document, 2000. URL `http://www.oss.sgi.com/projects/Pro64`. This is the website for downloading SGI's suite of optimizing compilers for the IA-64 platform. The suite includes C, C++, and Fortran90/95 compilers which conform to the IA-64 Linux ABI and API standards.

[18] Cygnus Solutions. Linux / IA-64 Developer's Release of the GNU Toolchain for the IA-64 Platform. Web document, 2000. URL `http://www.cygnus.`

com/ia64. This is the website for downloading Cygnus's Gnu toolchain for the IA-64 platform, including GCC, GDB, and the Gnu assembler.

[19] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000. ISSN 0018-8670. URL http://www.almaden.ibm.com/journal/sj/391/suganuma.html.