A TASK OPTIMIZATION FRAMEWORK FOR MSSP

BY

RAHUL ULHAS JOSHI

B.E., University of Pune, 2001

THESIS

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

Urbana, Illinois

Abstract

The Master/Slave Speculative Parallelization paradigm relies on the use of a highly optimized and mostly correct version of a sequential program, called *distilled code*, for breaking inter-task dependences. We describe the design and implementation of an optimization framework that can create such distilled code within the context of an MSSP simulator. Our optimizer processes pieces of Alpha machine code called traces and is designed to optimize code while adhering to certain restrictions and requirements imposed by the MSSP paradigm. We describe the specific places where our optimizer is different from that in a compiler and explain how the optimized traces are deployed in the simulator.

Table of Contents

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Overview 2
1.3 Outline 3

Chapter 2 Front End **4**
2.1 Issues in Representing Machine Entities 4
2.2 Representing Registers and Instructions 5
2.3 Handling Multiple Task Entries 7
2.4 MSSP Specific Instructions 8
2.4.1 Finding Task Liveins 8
2.5 Function Calls 9
2.6 Further Enhancements 10
2.7 Optimization 10

Chapter 3 Instruction Selection **12**
3.1 Instruction Selection Trees/Forest 12
3.2 BURG Pattern Matching 15
3.3 Virtual IR 16
3.4 Extracting MSSP Specific Information 17
3.5 SSA ϕ Elimination 17
3.6 Pre-allocation Optimizations 18

Chapter 4 Register Allocation **21**
4.1 Linear Scan Register Allocation 21
4.1.1 Preallocation 21
4.1.2 Live Intervals 23
4.1.3 Linear Scan Algorithm 24
4.2 Rewrite and Code Cleanup 25

Chapter 5 Deployment **27**
5.1 Generating Alpha Instructions 27
5.2 MSSP-Specific Annotations 28
5.3 Fixing Function Calls 28
5.4 Results 29

Chapter 6 Conclusion **33**

References **34**

List of Abbreviations

MSSP Master/Slave Speculative Parallelization

LLVM The LLVM compiler framework

IR Intermediate representation

CFG Control flow graph

VIR Virtual IR – IR using virtual registers

SSA Static single assignment

Chapter 1

Introduction

The Master/Slave Speculative Parallelization [1] (MSSP) paradigm relies on the use of an approximate version of a sequential program, called the *distilled program*, to parallelize its execution. The execution of a program is broken into several tasks and the distilled program acts, in essence, as a value predictor for values live across task boundaries. Since the execution speed a program in MSSP is determined by the speed of the distilled program, it is important to optimize the distilled code to exploit the new opportunity created by MSSP: the distilled code can sacrifice some correctness in favor of speed. This thesis describes the design and implementation of an optimization framework for optimizing such distilled programs.

1.1 Motivation

In MSSP the execution of a sequential program is broken into several tasks and inter-task dependences (values live across task boundaries, i.e., computed by one task and used by another task) are dealt with by predicting them instead of waiting for the previous task to compute those value. This prediction breaks the inter-task dependence and allows MSSP to execute the two tasks in parallel. To deal with incorrect predictions, at the end of each task MSSP checks if the predicted-liveins match the actual values of the liveins. If the check succeeds the task is committed else it is squashed and restarted. If the prediction accuracy is high, the execution speed of the program depends upon the speed at which predictions are computed. The novel feature of MSSP is that instead of using a hardware predictor for generating task livein predictions, it uses a piece of code to generate predictions. This piece of code is a simplified form of the sequential program under execution and is called the distilled code. Since the distilled code is used only as a predictor, it need not be correct. Hence MSSP can optimize it aggressively, eliminating computation that does not affect task liveins, and simplifying it so that task liveins are correct most of the times, but not always. Thus, MSSP can optimize the distilled code even when violating correctness in some cases. Such extreme optimization is called *approximation* or *distillation* and enables MSSP to create high-speed and high-accuracy predictors that are tailored to the specific program under execution. The MSSP paradigm is described in more details in [1].

Distillation can exploit many different forms of program behavior to create distilled code. One general technique is to focus on the frequently executed sections of the code for distillation. While a profile-guided compiler optimization also uses the same technique, it is constrained by the fact that although it can slow down the computation along a cold path arbitrarily, it cannot transform

the code so that the execution of a cold path leads to incorrect computation. As opposed to that, distillation can completely neglect the cold paths in a program. The resulting code, consisting of only the hot paths in a program, can have many redundancies and inefficiencies that can be eliminated using standard compiler optimizations techniques. The first step in the process, extracting hot sections of code and deciding task boundaries, is called *program orienteering* and is described in more details in [2]. Here, we describe the design of an optimization framework that can optimize such hot sections of code which are called distilled *traces* or just traces[1].

1.2 Overview

The current program orienteering framework operates in the context of an MSSP simulator and works on program binaries compiled for the Alpha platform. We had three choices for the design of the optimizer:

- Implement the approximating optimizations for an intermediate representation that is essentially the Alpha instruction set. This was the approach followed in [1]. The advantage of this approach is that it can perform aggressive platform specific optimizations. Also, standard compiler optimizations can be "approximation enabled," so that they can optimize the program incorrectly if it helps speed up the program without introducing too many incorrect predictions. The disadvantage is the effort involved in implementing a large number of compiler optimizations.

- Use an existing link-time or post-link time optimizer for Alpha like OM [3] or Alto [4]. Although such tools provide a higher level IR than the native instruction set, their IR does not maintain certain MSSP specific information (like the mapping between the PC of some instructions in a trace and their PC in the original sequential code), and hence cannot be used easily in the context of distillation.

- Use an existing compiler framework which implements many standard compiler optimizations but expose to it just the hot portions of the code. This approach makes available a large number of optimizations and in many cases hiding the cold parts of the code from the compiler itself creates many opportunities for optimizations like dead code elimination, redundancy elimination, constant propagation, and many others. The disadvantage of this approach is that it requires the translation of the trace code into the compiler's IR and the translation of the optimized IR back to machine code. Also, optimizations cannot be "approximation enabled." However, this a quick and relatively easy way to build an optimizer as compared to a ground up implementation.

Considering the above choices we decided to reuse an existing compiler framework. We wanted to use a compiler with an IR that was close to Alpha's RISC instruction set. The LLVM compiler framework [5] provides such an IR and a number of useful features as well. Hence we decided to use it. Our optimization framework translates Alpha instructions into LLVM instructions and optimizes the resulting LLVM program using optimizations built into LLVM. Finally, a back-end generates Alpha instructions that form the distilled code. The main phases that form the optimizer are:

[1]The difference between a task and a trace is that a trace can contain multiple tasks and task boundaries.

1. Front end – The front end translates the trace code into the LLVM IR, trying to create a faithful representation of the trace code using the LLVM IR.
2. Optimize – This phase optimizes the resulting LLVM code using LLVM optimizations. In our implementation we apply two simple but effective optimizations: Control Flow Graph (CFG) simplification and dead code elimination. The aim is to eliminate computations that don't affect the values of task liveins for that task.
3. Instruction selection – The first step in translating the optimized LLVM IR back to Alpha machine instructions is to select instructions to implement each IR instruction or group of IR instructions. This phase, as in a general compiler, assumes an infinite set of virtual register at disposal.
4. Register allocation – The set of virtual register used during instruction selection need to be mapped to the finite set of registers available on the Alpha. In addition, at task boundaries, the task liveins must be in correct registers and unlike a register allocator in a compiler here the allocator cannot spill any virtual register to memory. This phase also does simple cleanup of the code to eliminate redundant instructions.
5. Deployment – Finally, the optimized code for a trace must be deployed within the simulator for execution.

1.3 Outline

Chapter 2 describes the front end of the optimizer and explains how it resolves the conflicts between the LLVM IR and the trace machine code. It also explains how we choose to represent and retain certain MSSP specific information that is required by the later phases of the optimizer. Chapter 3 describes our tree pattern-matching instruction selector and how it extracts the MSSP specific information from the LLVM IR. Chapter 4 describes the implementation of an efficient register allocation algorithm called linear scan [6, 7] and the changes required to the algorithm to satisfy the requirements on the task code imposed by MSSP. Chapter 5 describes the deployment of the optimized traces in the simulator and presents some statistics about the code generated by the optimizer while optimizing traces in a subset of SPEC2000 integer benchmarks. Finally, chapter 6 concludes the thesis.

Chapter 2

Front End

The front end of the optimizer translates Alpha machine instructions to LLVM IR. Apart from representing instructions and registers using LLVM, the front end also has to represent information from the machine code that is specific to MSSP. This has to be done is such a manner that the back end can extract the information for code generation irrespective of the optimizations applied. This chapter explains how our front end works and how we choose to retain MSSP specific information in the IR.

2.1 Issues in Representing Machine Entities

The front end is responsible for creating a faithful translation of the trace code and for that purpose it has to take care of the differences between the machine code and the LLVM IR structure and assumptions. The following are the main issues in representing various pieces of machine code using LLVM IR:

- Machine registers – LLVM uses an IR that is in Static-Single Assignment (SSA) form [8]. This means that each variable in the program has a single static definition. Secondly, these variables are typed. The trace code, on the other hand, uses machine registers and as such is not in SSA form. The registers here can have one of the two types corresponding to integer and floating point registers. An integer register can be used to hold integers of various sizes, characters, and boolean values. The front end either has to analyze the code to deduce the type of the value in a register or make conservative assumptions when translating the code.

- Machine and MSSP specific instructions – Apart from containing instructions that can be represented directly using LLVM instructions, the trace code can also contain certain instructions that cannot be directly represented in the LLVM IR. Examples include the Alpha UMULH instruction that computes the upper 64 bits of the product of two 64-bit integers. The optimizer is required to handle these instructions appropriately and represent them in a suitable manner.

- Multiple task entries – The code to be optimized for MSSP can, in general, have multiple task boundaries in it and, hence, can have multiple entry points. LLVM, on the other hand, requires that each piece of code (a Function) has a single entry point (it can have multiple exit points). The front end should therefore "merge" all the trace entries into a single entry.

- Register assignment at task boundaries – The distilled code, in general, can have a different register allocation than the original program. In MSSP the communication of task liveins from the predictor to the task in question is done using the register file, i.e., at each task boundary the register file of the processor executing the distilled code is copied and used as a starting point for the execution of the next task. This implies that the distilled code should not only compute the values of task liveins but at task boundaries these values must reside in the registers that the next task expects them to be in. This can be done either in hardware using a special functional unit to permute the register file appropriately or in software by assigning task liveins appropriate registers. The hardware approach is expensive, and software approach is feasible because register pressure in distilled code is less than that in the original code.

In the following sections we describe how the front end addresses each of these issues.

2.2 Representing Registers and Instructions

As mentioned above, there are two issues in representing registers: types and multiple definitions. Our front end takes a conservative approach and uses the coarse type information (integer vs. floating point) that is readily available in the machine code. Thus, each machine register is converted to either an LLVM `ulong` or `double` variable. The second issue, eliminating multiple static definitions, is essentially the process of converting from a non-SSA IR to SSA, and an algorithm for this is described in [8]. LLVM already provides an implementation of this algorithm which we choose to use. The variables to be converted into the SSA form need to be allocated on a stack using LLVM `alloca` instruction. LLVM's memory to register promotion pass (`mem2reg`) can then convert these stack allocated variables into variables in SSA form and insert SSA ϕ nodes at appropriate points in the code. Thus, for each trace, the front end `alloca`tes 31 `ulong` and 31 `double` values on the stack. During translation of individual instructions code is generated to load the input registers of an instruction from this stack, compute the output value, and store it on the stack. At the end of the translation the `mem2reg` pass eliminates these load/store instructions.

Translation of many individual instructions is straightforward because of the existence of a corresponding LLVM instruction. We summarize how the front end handles each type of Alpha instruction here:

1. Integer and floating point arithmetic – These instructions can be translated directly into their LLVM counterparts. Many of these instructions have versions corresponding to quadword, word, half-word, and byte-level operations, or single- and double-precision operations. Since the front end uses coarse types for representing registers, when translating such instructions, it generates code to first load a 64-bit value from the stack and then cast it to the appropriate type. This approach is simple to implement and does not interfere with dead code elimination.

2. Load/Store – These are also translated to LLVM `load/store` instructions. LLVM versions of these memory operations take a pointer operand whereas the Alpha versions compute the address by adding a constant to the value of an integer register. The front end translates these instructions into an addition, followed by a `cast` instruction to create a pointer of the appropriate type, and finally an LLVM `load/store`.

3. Branches – Conditional branches are translated into a comparison to compute the branch predicate followed by an LLVM `br` instruction.

4. Indirect branches – Indirect branches are represented using the LLVM `switch` instruction. Each possible target of the indirect branch is assigned a unique integer. The front end defines the `llvm.alpha.jump` intrinsic that accepts the address register for the indirect jump as the input argument and returns the unique integer associated with that target. This return value is then used as the `switch` predicate. In addition, MSSP requires that the original PC be associated with the target of any indirect branch. Therefore, the front end adds the `mark_addr` intrinsic at the start of each basic block that is a target of this `switch` instruction.

5. Machine specific instructions – Certain Alpha instructions have no direct and easy mapping to the LLVM IR. One option is to represent such instructions by complex pieces of LLVM code. For example, a CTPOP instructions counts the number of set bits in a registers. Although it can be converted into a loop in LLVM, it would be very difficult for the back end to recognize such loops and regenerate the CTPOP instructions. Secondly, such detailed translation is not required for optimizations like dead code elimination. Hence, we chose to represent such instructions in a simpler manner using function calls. These function calls can be easily recognized by the back end and translated to a single instruction. The LLVM compiler framework has the ability to define *intrinsic* functions: functions about which the compiler knows nothing and hence makes conservative transformations. One drawback of using such intrinsics is that they interfere with the dead code elimination process: because of conservative assumptions, LLVM cannot eliminate such function calls. Specifically, these functions can, in theory, write to memory and hence cannot be eliminated. LLVM plans to provide some way of specifying which intrinsics write to memory and which don't. With that, these intrinsics no longer interfere with the dead code elimination process. Table 2.1 shows some of the intrinsics we defined and the type of instructions they are used for. The `llvm.alpha.unimp` intrinsic is used to represent any non-control-transfer instruction for which the front end has not defined a translation. This intrinsic enables reallocation of the registers used and defined by the un-handled instruction without knowing the semantics of the instruction.

Intrinsic Name	Purpose
`umulh`	Upper 64 bits of the 128 bit product of two integers
`vecop`	Alpha vector-like instructions like MINU, CMPBGE
`read_gpr/read_fpr`	Register read at task entry
`write_gpr/write_fpr`	Register write at task boundary/function calls
`verify`	Represents MSSP VERIFY instructions
`unimp`	Non-control transfer instructions not handled by the front end
`unimp_fix`	Unimplemented instruction for which registers cannot be renamed

Table 2.1: Intrinsics used by the front end to represent special instructions

6. Function calls – A function call can essentially be treated as a single instruction that uses and defines multiple registers. In the most general case its an instruction that can read all the registers and define all of them. Secondly, unlike un-handled instructions, the code executed by the function call is not available and hence registers cannot be renamed according to the register allocation of the code being optimized. We explain how we handle function calls in section 2.5.

Before proceeding, we present an example of how a snippet of Alpha machine code is translated into the LLVM IR. Figure 2.1 shows the translation of a simple Alpha code snippet that loads a pointer and then increments it.

a)

```
ldq   t1,0(a0)
addq  a0,8,a0
```

b)

```
%a0 = alloca ulong
%t1 = alloca ulong

;translation for ldq t1,0(a0)
%a0v0 = load ulong* %a0
%add.0 = add %a0v0,0
%cast.0 = cast ulong %add.0 to ulong*
%ld.0 = load ulong* %cast.0
store ulong %ld.0,ulong* %t1

;translation for addq a0,8,a0
%a0v1 = load ulong* %a0
%add.1 = add %a0v1,8
store ulong %add.1,ulong* %a0
```

Figure 2.1: An example of translation of Alpha code to LLVM: a) the Alpha code, and b) its translation to LLVM

2.3 Handling Multiple Task Entries

LLVM requires that each function has a unique entry point. In the case of MSSP tasks this may not be the case. In general, each task boundary in the code forms an entry point into the distilled trace. The front end represents multiple task entries by creating a dummy entry block for each trace and adding an LLVM `switch` instruction to jump to the basic blocks that start at the task entry points. Effectively, each entry point in the trace is assigned a number, and the LLVM function, on entry, finds which task entry to execute using the `find_entry` intrinsic and branches to that entry. The back end can recognize such dummy entry blocks (by looking for the `find_entry` intrinsic) and ignore them during code generation.

When control enters the distilled code along one of the task entry points (usually after a misspeculation), the register values have been already set as a part of the misspeculation recovery and the distilled code is expected to use these values. Thus, each task entry is an intrinsic definition of all the registers live at that point. To model this fact, we use two LLVM intrinsics, `read_grp` and `read_frp`, that are defined to read the value of an integer and floating point register respectively. The single argument to these intrinsics specify which register to read. The front end uses these intrinsics to set the values of the stack allocated registers at each entry point in the distilled code. Thus, each task entry reads the values of all the registers using these intrinsics. This captures the fact that all task entries are definitions of the machine registers.

The optimizer also needs to remember the original addresses of task entry points so that during deployment of the optimized code it can appropriately fix up branches. For this purpose, at each entry point a `mark_addr` intrinsic call is added. The single 64-bit parameter to this intrinsic is the address of the entry point.

2.4 MSSP Specific Instructions

MSSP extends the Alpha instruction set with instructions to manage the creation and verification of tasks and an instruction for profiling. The FORK and VERIFY instructions are used at task boundaries. A FORK creates a new task and continues execution in the distilled code on the master and in the verify block associated with the FORK on the slave processor. Since LLVM does not provide such control transfer instructions, the front end models a FORK using a conditional branch, with the next instruction following the FORK as one target, and the verify block as the other target. The predicate of this branch is the output of the `fork` LLVM intrinsic. That way, the back end can distinguish between these conditional branches and other conditional branches and generate appropriate code. The verify block can contain arbitrary code and terminates with a VERIFY instruction that starts the execution of the task on the slave. From the point of view of distilled code, the VERIFY is effectively an exit from the code, so it is represented using a call to the `verify` intrinsic followed by a return from the function. As mentioned above, at the end of a verify block, the task liveins must be in the appropriate registers. The front end notes this fact by using another pair of intrinsics: `write_gpr` and `write_frp`. These intrinsics take two parameters: an `ulong` or `double` value and a register number. The semantics is that the input value is written into the specified register.

The register read and write intrinsics inserted at task entry and verify points accurately capture the registers live-in to a task at its entry and liveout to a task at its exit. They also serve as register allocation hints for the register allocator. Specifically, if a particular value, say $v0$ is used as a parameter to a register write intrinsic that writes to register $r3$, it may be beneficial to allocate $r3$ to $v0$. We describe the register allocation process in more detail in chapter 4.

The MASTER_END instruction is used to terminate the execution of the program in MSSP mode and continue in a sequential mode. Again, this is effectively an exit from the distilled code. However, unlike a verify block, none of the register values at the MASTER_END will be used for predictions and hence all registers are dead at a MASTER_END. The front end models this using an `master_end` intrinsic followed by a return. However, the register write intrinsics are not used here because all registers are dead.

The current MSSP simulator also uses a PROFILE instruction to collect information about the behavior of the program. This instruction is used currently to collect an edge profile and identify the cold sections of the code. As far as the program visible processor state is concerned, these profile instructions do not affect any of the registers and do not lead to any control transfer. Thus, the front end chooses to represent them with a simple `profile` intrinsic (with arguments specific to the profile instructions). This intrinsic neither defines not uses any registers and hence does not interfere with dead code elimination.

The program orienteer may also create special blocks that are tagged as defining cold paths. These blocks have an unconditional branch at their end and they represent cold parts of the code at that point. Since the aim of the optimizer is to hide such cold paths during optimization, these block are translated just as other blocks so that they branch to some other hot block in the trace.

2.4.1 Finding Task Liveins

For each VERIFY instruction the front end needs to know the registers live at that point. This information can be computed either by doing an interprocedural register liveness analysis [4] or profiling the liveins. When profiling the liveins, the distiller can deploy an unoptimized version of

the trace for some time and have the MSSP verification unit record the liveins for each task. The advantage of the these profiled liveins is that they are relatively simpler to implement than a whole program register liveness analysis and are much more precise[1]. Secondly, the profiling approach also allows the frequency (or "hotness") of each livein to be profiled. Thus, a livein to a task may have an exposed use along an infrequently executed path. In such cases, it may be advantageous to ignore that livein if it helps in optimizing the trace.

The profiled liveins approach can be used either statically or dynamically. In the static approach the simulator is run once to collect the liveins for each task and a static table of the task liveins is constructed. This table is then used to find the liveins for each VERIFY instruction. This is the approach currently taken in the optimizer. The second approach would be to have the verification unit collect the livein information for some time and then use that information to optimize traces during the same run. We plan to implement this approach in future.

2.5 Function Calls

The register read and write intrinsics can also be used to translate function calls. In general, the function called by the function call may not be available for optimization or the code may be unknown. In the most conservative case, when the callee code is not available for analysis, the front end has to assume that the called function can read and write all registers during the function call. This means that, similar to a task boundary, the values that are liveins for the called functions must be available in proper registers. Similarly, after returning from the function, the situation is similar to a task entry and the return values from the function calls must written to the stack allocated registers. There are three choices in deciding which registers to write before a function call and which registers to read after a function call:

1. All registers – As stated above, this is the most conservative choice. However, it has the advantage of simplicity and it also helps guide the register allocation process, as we will discuss in Chapter 4.

2. Calling convention – A more practical approach is to assume that the code complies with the Alpha calling convention. In that case, before the function call, the front end can add write intrinsics for only those register that a compliant callee can read, viz., the argument registers `a0 - a5`, saved registers `s0 - s5`, `sp`, `fp`, `at`, and `gp`. During the function call, the callee may save and restore some registers. These registers have the same value in the caller before and after the function call and hence there is no need to add read intrinsics for them. Thus, as per the Alpha calling convention, the front end can add read intrinsics for `v0`, `at`, and `t0 - t11`. Note that this still adds a large number of register read intrinsics after the function call. However, it is expected that many of these read intrinsics will be dead, since the registers they read are dead after the function call. In particular, all the temporary registers would be dead at the function return point. The front end could also aggressively assume that this is the case and do not insert read intrinsics for temporaries.

3. Interprocedural register liveness – A more efficient approach can be used when the callee code can be analyzed. This can give information about the exact registers the callee reads and defines and the front end needs to add write/read intrinsics for only these registers.

[1] Registers statically livein to a task along a path that was never exercised when the liveins were profiled will not be in the profiled livein set. Thus, profiled liveins are potentially speculative.

4. Interprocedural optimizations – An even more powerful way to handle function calls would be to optimize the caller and the callee together and apply inter-procedural optimizations like register allocation.

Our front end currently uses the most conservative strategy stated above. It is simple to implement and helps register allocation. The actual function call instruction is translated into an `llvm.alpha.unimp_fix` intrinsic. This intrinsic is used to represent instructions whose register usage cannot be changed (i.e., they use and define a fixed set of registers). The single parameter to this intrinsic is the bit pattern of the instruction, which is used to regenerate the instruction during code generation.

A function return also needs to have some register write intrinsics before it. Generally, the return instruction is a user of register `ra` which holds the return address. Secondly, function return values (`v0`, `f0`, `f1`), the saved registers (`s0 - s6`), and `gp` are live at the return. The front end adds register write intrinsics for all these registers before the return.

2.6 Further Enhancements

Our front end is quite conservative currently, and can be extended to perform a more accurate translation of the machine code into LLVM IR. Two possible improvements are:

- The front end can analyze the machine code to deduce more accurate type information for registers. The instructions using a particular register can serve as a guide in this type deduction process. For example, if an integer register is always used in longword arithmetic (ADDL, LDL, STL etc.), the front end can deduce its type as `int`. This would eliminate many cast instructions that are needed when all registers are assumed to be 64-bit integers. The front end has to take care of cases such as the same register being used to hold a long integer in one section of code and a boolean in another section of code. This can be done if unrelated uses of a register are segregated by creating "webs" as in register allocation [9].

- The front end can analyze the stack usage in the code to find stack allocated local variables and their types. These can be directly converted into LLVM `alloca` instruction. This could enable the optimizer to eliminate instructions that define stack allocated locals that are dead in the distilled code. Note that MSSP requires that the stack frame layout and stack frame size of the distilled code be the same as that of the original code. This is because a task livein may reside in memory, in which case the distilled code should write the predicted value to the correct memory location. Thus, the stack space allocated to dead locals cannot be eliminated but the memory instructions that write to them can be.

Such enhancements can help in improving the effectiveness of the optimizations that are applied to the distilled code.

2.7 Optimization

As explained in this chapter, the front end translates the trace into the LLVM IR using stack allocated registers. The first step in the optimization process is to translate this non-SSA code into SSA form. The LLVM `mem2reg` pass is run on the code resulting from the translation of individual instructions. This pass promotes all the stack allocated "registers," and eliminates the

load/store instructions created during the translation of individual instruction. When a use of a register is reached by multiple definitions, SSA ϕ nodes are inserted at appropriate places in the CFG. Next, the CFG simplification pass is run on the resulting code. When translating each instruction individually, the front end creates separate basic block(s) for each instruction. The CFG simplification pass simplifies the CFG by merging straight line chains of blocks into bigger basic block. At this stage, the code is passed through the dead code elimination pass. This pass eliminates dead computation that does not affect the values of task liveins (which are marked in the form of write intrinsics at verify points). After these optimizations have been applied the code is much simpler than the original code and is ready for translation back to Alpha instructions.

Chapter 3

Instruction Selection

This chapter explains the first step in translating the optimized LLVM IR back to Alpha instructions: selecting Alpha instructions to implement each instruction in the IR. This process is called *instruction selection*. An instruction selector can either generate a one-to-one translation of each IR instruction or can take advantage of the machine ISA to translate multiple IR instructions into a single machine instruction. As an example, a load from memory followed by an address increment can be translated into an auto-increment load, if such an instruction is available. In general, we can define patterns in the IR that correspond to each machine instruction and then do a pattern matching of these patterns against the input IR. Whenever a pattern for an instruction is matched, the corresponding machine instruction can be generated. We use this pattern matching approach in our instruction selector and use a tool called BURG [10] for constructing the instruction selector. To avoid complexity, the instruction selection phase of the optimizer assumes the existence of an infinite set of virtual registers for code generation. Thus, each instruction generated by the selector is assigned a new output register. The register allocation phase then maps these virtual registers to physical registers.

3.1 Instruction Selection Trees/Forest

The *instruction selection tree* is a data structure created from the IR to ease the process of pattern matching. It is essentially the DAG [9] of each basic block in the IR, split at certain points to create a tree. Specifically, if a DAG node n has 2 or more parents, then the DAG is split at n and a subtree rooted at n is created. For each parent of the node n, a "representative" node is added to its tree to represent the subtree rooted at n. The reason for splitting the DAG is to create trees from the DAG, which is a prerequisite for BURG. Using the instruction selection tree instead of the DAG for pattern matching has the advantage of simplicity but prevents pattern matching that spans across common subexpressions. LLVM provides the necessary data structures for this process, and given a function, creates a set of IR trees for each instruction (an instruction forest). If two instructions, say I1 and I2, are in the same basic block such that I1 defines a value that I2 uses and I2 is the only user of the value defined by I1, then the tree for I1 is added as a child of the root node in the instruction selection tree for I2. In general, an instruction can be a user of an unbounded number of values, and the instruction selection tree can have an arbitrary arity. BURG however requires that the input trees be ordered binary trees, with pointers to left and right children. Under this restriction, a user of more than 2 values can be represented by either

(i) using a binary tree of unbounded depth, or (ii) using a special node that represents a list of uses. The second approach is simpler to deal with and is followed by the instruction selection tree generator of LLVM. As an example, Figure 3.1 shows an LLVM basic block and its instruction selection forest. Note that the leaf nodes of the instruction selection tree are either constants, or values with multiple uses, or/and values defined outside the block. Similarly, the root of each tree is either a value with multiple uses, or no uses (for example, the `br` instruction), or a single use outside the block. Once such a tree is constructed, BURG allows us to define pattern(s) for

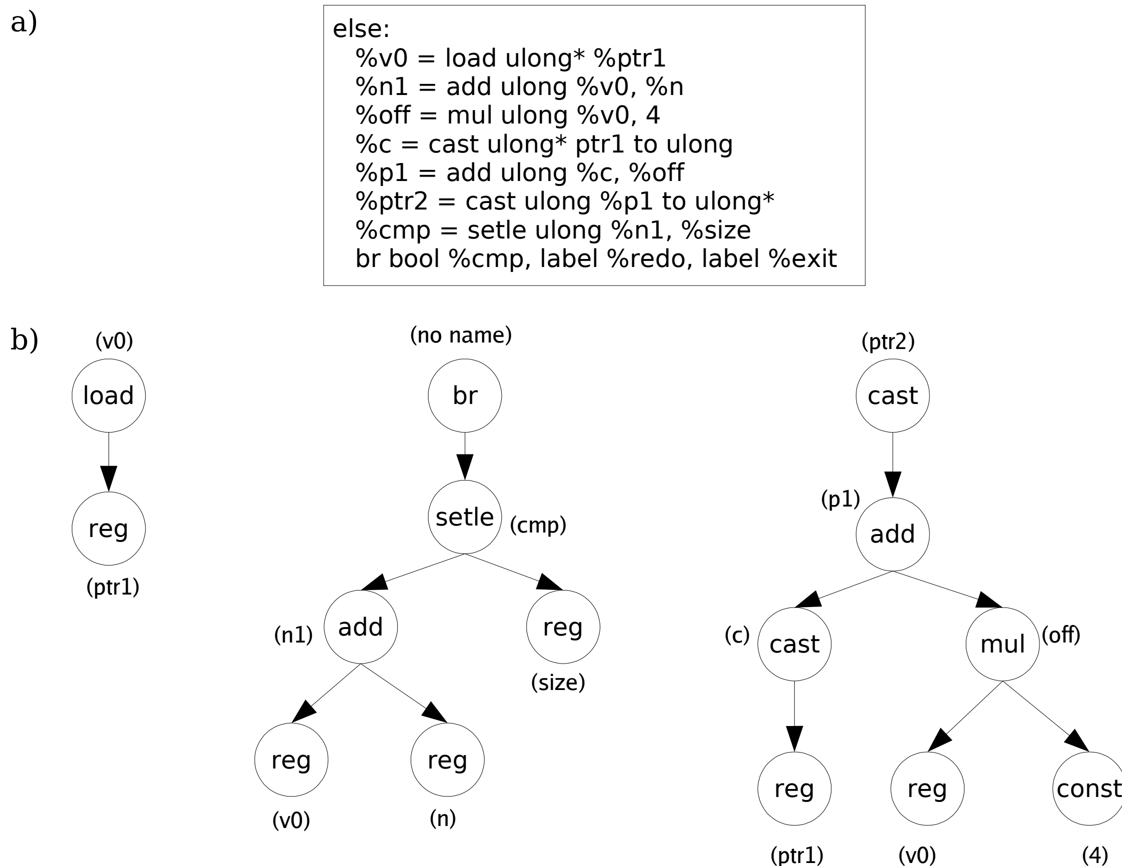

Figure 3.1: An (a) LLVM block and (b) its instruction selection forest

each machine instruction in the form of smaller trees similar to the trees in the instruction selection forest and then it can do a pattern matching of these tree-patterns for instructions against the input instruction selection tree. Essentially, the patterns can be viewed as defining a grammar for a tree, and the pattern matching process as parsing an input tree against this grammar. Therefore, this tree-pattern matching is also called *tree parsing* and the patterns are defined in the form of a *tree grammar*. As an example, Figure 3.2 shows how the the second tree in Figure 3.1(b) can be parsed against a simple tree grammar and the code generated from the pattern matching. Figure 3.2(a) shows the tree grammar and the associated instructions. For each rule, the left hand side of the rule is assigned number 0, and the non-terminals on the right hand side are assigned numbers starting from 1. Figure 3.2(b) shows the parse of an example tree, with the bold numbers indicating the rules against which the subtree matches, and also shows the instructions generated for this tree

during the process of parsing. It can be seen that the for correct code generation, the instruction

a)

```
1) tree := Branch(cmp)          bne $1.reg
2) cmp := SetCC(reg, reg)       $0.reg = cmpCC $1.reg, $2.reg
3) reg := quad                  $0.reg = $1.reg
4) reg := Rep                   $0.reg = Reg assigned to tree root of Rep
5) quad := Add(reg,reg)         $0.req = addq $1.reg, $2.reg
```

b)

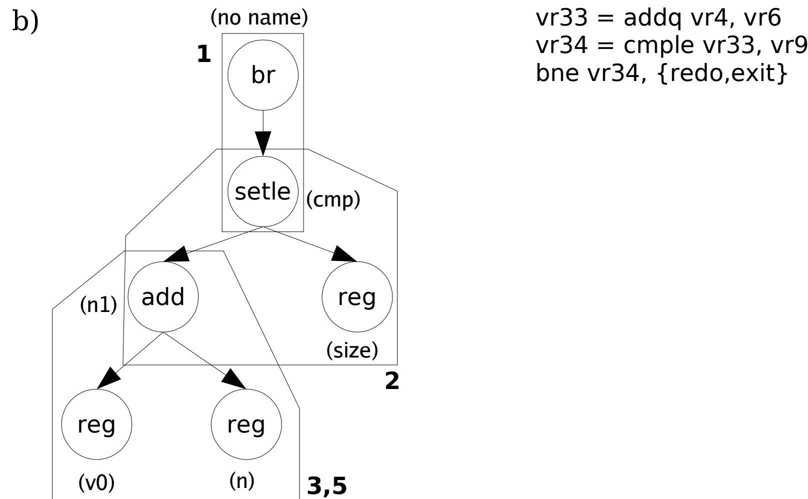

Figure 3.2: Code generation using tree parsing: a) the tree grammar, and the associated instruction selection code, and (b) an example parse of a tree, and the generated code.

selection tree must be traversed post-order (i.e., bottom-up), so that the code for all the children of a node is generated before the code for that node (if any) is generated. Secondly, the tree parser needs to maintain certain attributes for each tree node and pass them to its parent. In our example, the attribute of interest is the (virtual) register holding the output of an instruction. Also, at the end of the parsing of a tree, the attribute of the root node may need to be preserved if that tree has representative nodes in some other trees. In the next section, we describe a tool called BURG that helps in writing such a tree parser.

Before proceeding, we make two observations about the tree-parsing process. The first is that the children of a node in a parse tree may not be its immediate children. As an example, if the ISA defined an instruction that compared two registers and branched based on the result, we would replace rule 1 in the grammar in the example above with the rule tree := Branch(SetCC(reg,reg)). In that case the children of `br` are the nodes `n1` and size, rather than the node cmp. Second, the same node can have different children in different parse trees. Again, if in the above grammar, we choose to retain the original tree := Branch(cmp) rule, the tree now has two parses. In one parse, using rule 1, the `br` node has a single child, cmp, as shown in the figure. In the second parse, using the new rule, the `br` node has two children, `n1` and size. In general, the children of a node in the parse tree depends upon the grammar rule associated with that node in that parse.

3.2 BURG Pattern Matching

BURG [10] is tree-parser generator, similar to yacc. It accepts a cost-augmented tree grammar, and generates a tree-parser for parsing trees using that grammar. In general, the input tree-grammar may be ambiguous, so that for a given input tree, 2 or more parses exist. This is a common situation in instruction selection, so that a given piece of IR can be implemented using many different sets of instructions. In that case, we would like to generate code with minimum "cost," where the cost is a user defined metric associated with each instruction. BURG allows each grammar rule to be augmented with these costs. When multiple parses of a tree exists, it chooses the parse with the minimum total cost. This parse is called an *optimal parse* and can be found in linear time.

BURG requires that each input tree node be associated with an *operator* that identifies the type of the node. For nodes that represent instructions, this is normally the operator for that instruction. In addition, some more operators are defined for nodes representing constants, tree representatives, and use-list (used for nodes with more than 2 operands). Each grammar rule is assigned an *external rule number* that identifies the rule. BURG accepts a file that specifies the tree grammar and generates C code that contains helper functions which can be used to write a parser. This approach allows us to add code for creating instructions and attributes at proper places during the parsing process. The parser traverses each instruction selection tree twice. The first traversal is a top-down traversal, called *labeling*, and the second, called *reduction* is a bottom-up traversal during which the actual code generation takes place. BURG provides the following functions that help in writing a parser (more details can be obtained in [10]):

- `burm_label` – This function implements the first pass of the parsing process, and is called on the input tree at the start of the process. It returns 0 if the input tree cannot be parsed according to the grammar. This is the only function required to implement the labeling pass.

- `burm_rule` – The second pass, the bottom-up traversal (or reduction), is generally implemented as a recursive procedure similar to a recursive post-order traversal of a binary tree. As mentioned before, the *reducer* needs to find out the children of the current node based on the grammar rule associated with the current node. The `burm_rule` function is provided for exactly this purpose. It finds the *external rule number* of the grammar rule used to reduce the input tree node.

- `burm_kids` – This function is used to find the actual children of the node. It accepts the external rule number found using `burm_rule` and the tree node, and returns a list of all the children of that node in the parse tree. Once we have these children node the reducer can recurse on each of them.

Our instruction selector defines a simple grammar that directly corresponds to the translation of each Alpha instruction to LLVM IR. This is feasible because we apply only simple optimizations on the generated IR. A more complex grammar would be needed if optimizations like constant folding and common subexpression elimination are applied. In general, it is useful to extract common "sub-patterns" from this grammar and define a grammar rule for them. When such a sub-pattern is matched, no code is generated but an attribute value is computed and passed to the parent. Actual instructions are generated when one of the "top-level" patterns match. Figure 3.3 show the high-level algorithm of the instruction selector based on the functions provided by BURG. It is basically a post-order traversal of the instruction selection tree with additional code to generate

and propagate attributes from children to their parent. In function Reduce, the algorithm first visits all the children and collects their attributes in an array. Then, it visits the root node and uses the attributes of the children to generate code for the root. As mentioned before, this "code generation" can result in the generation of an actual instruction (in case of top-level patterns) or can result in the computation of the attribute of the root using the attributes of the children, without generating any actual instruction (in case of sub-patterns).

3.3 Virtual IR

The instruction selection phase generates Alpha instructions that use both physical registers and virtual registers. We call such instructions *virtual IR* (VIR). In addition, branches in VIR are not fall-through as in case of the actual Alpha branch instructions, but specify both a true and false target. The advantage of this is that the instruction selector does not have to keep track of the actual layout of the generated code in the memory, which is required when fall-through branches are used. Secondly, this choice is in conformance with the LLVM IR and hence code generation for LLVM branches is simplified. Instructions in VIR can have one of the following types:

1. VInst – These represent all non-control-transfer instructions supported by the front end. Each VInst maintains a pointer to an actual Alpha instruction (from OPTIR). The input/output registers of this OPTIR instruction are initially set to `r31`. The actual input/output registers are assigned to the OPTIR instruction after register allocation. Thus, a VInst is an instruction whose input/output registers can be renamed as per the register allocation of the surrounding code.

2. VInstBranch – These represent conditional and unconditional branch instructions. It is a subclass of VInst and maintains a list of all the targets of the branch (including the fallthrough target).

3. VInstUnimp – These represent non-control-transfer instructions that the front end translates using the `unimp` intrinsic. These instructions maintain the original bits of the unimplemented instruction they represent, which are used during code generation. Although marked as unimplemented, register used and defined by these instructions can be renamed as per the register allocation of the surrounding code.

4. VInstUnimpFix – These represent instructions that the front end translates using `unimp_fix` intrinsic. Here again, the original bit pattern for the instruction is retained for code generation. However, registers used and defined during the execution of this instruction cannot be renamed (for example, function calls). Thus, these instructions use and define a fixed set of registers, and maintain two sets of registers: the register used and the registers defined during their execution. The front end inserts write and read intrinsics for these registers before and after the instruction respectively and these use/def register sets are also used during liveness analysis.

5. VInstUnimpBranch – These represent indirect branches and maintain the original bit pattern of the indirect branch. Here the register used by the indirect branch can be renamed as per the register allocation.

3.4 Extracting MSSP Specific Information

As mentioned in Chapter 2, the LLVM IR generated by the front end contains certain MSSP specific information that needs to be extracted by the instruction selector and communicated to the generated VIR. This information is mostly contained in the intrinsic function calls contained in the IR, and can be easily extracted. In the following, we describe the key MSSP specific information that is maintained and extracted from the IR:

1. Original Addresses – MSSP requires that the mapping between an instruction in the optimized trace and its address in the original sequential code be maintained in some cases. The key points in the trace code where such a mapping is required are: task entry points, FORK instructions, and any instruction that can be the target of an indirect branch. Whenever the front end finds such an instruction, it inserts the `mark_addr` intrinsic, with the original program PC as the parameter. The instruction selector, on finding this intrinsic, extracts the PC and annotates the VIR with this information. This annotation is mostly associated with basic block containing the `mark_addr` intrinsic. In certain cases however (like function calls), the original PC needs to be associated with the instruction rather than the block containing that instruction. For such mappings, the front end creates a table of the mapping between LLVM instructions and the original PC and the instruction selector looks up this mapping to extract original PC for instructions.

2. Task liveouts – The front end marks the liveouts of a task by inserting register write intrinsics before the `verify` intrinsic. These liveouts are physical registers and are live at the end of a block containing a VERIFY instruction. These liveouts are required for correct live variable analysis for register allocation. We choose to represent task liveouts by associating with each basic block in VIR a set of *liveout seeds*, which is the set of registers that are always live at the output of that block. This set is empty for all non-verify blocks. The liveness analysis uses these liveout seeds by defining the liveouts of a block B as

$$liveout(B) = \Big(\bigcup_{S \in Succ(B)} livein(S) \Big) \cup liveout_seeds(B)$$

   so that liveout seeds are always live at the end of the block.

3. Task liveins – The front end marks the registers live at a task entry by inserting register read intrinsics. These register read intrinsics are converted into register copy instructions by the instruction selector. These copy instructions are then used by the register allocator as register allocation hints, as described in Chapter 4.

3.5 SSA ϕ Elimination

The SSA ϕ instructions in the IR are eliminated by the instruction selector by inserting register copies in all the predecessors of the block containing the ϕ, in accordance with the semantics of the ϕ instruction. This can introduce redundant register copy instructions along some program paths. Specifically, if a predecessor p of a block b containing the ϕ is also a predecessor of some other block b', then the register copy is redundant on the path from the predecessor p to b'. Figure 3.4 shows an example. In such cases, the copy is really needed on the edge $p \to b$. Such an edge whose source

block has more than one successor and whose target block has more than one predecessor is called a *critical edge* [9] and needs to be broken by introducing a block along the edge $p \rightarrow b$ as shown in Figure 3.4(b). This can be done either in the VIR or in the LLVM IR. Since LLVM already provides a pass to break critical edges, we choose to do this in the LLVM IR.

3.6 Pre-allocation Optimizations

After ϕ elimination, the instruction selector applies two optimizations to simplify the code before register allocation. The first optimization eliminates empty unconditional branch blocks that might have been introduced in the CFG during critical edge breaking. Such blocks are eliminated by modifying all their predecessors to jump to the only target of the unconditional branch contained in such blocks. The second optimization, *hoisting*, tries to create empty unconditional branch blocks that can be eliminated by the first optimization. Specifically, it finds a block (b) with two successors, one of which (m) has an MASTER_END instruction, and the other b' is terminated by an unconditional branch to a FORK block, and tries to move all the instructions from b' to b past the conditional branch that terminates b. Such code motion is valid because none of the registers are live at the MASTER_END. If the optimizer can move all the instructions from b' to b, then b' becomes an empty unconditional branch block and can be eliminated (see Figure 3.5). This optimization is able to eliminate some of the non-empty blocks that were introduced in the CFG during critical edge breaking.

```
// This function accepts an instruction selection tree and generates code that uses virtual
// registers. It returns the attribute for the root node, which generally is the virtual
// register holding the output of the instruction generated at the root node.
TreeAttribute InstructionSelect(InstrTreeNode root):
    // The first pass: a top-down traversal
    burm_label(root)

    // The second pass: a bottom-up recursive postorder traversal
    return Reduce(root)


// This function traverses the instruction selection tree bottom-up and in the process
// generates code and/or propagates attributes from the children of a node to that node.
TreeAttribute Reduce(InstrTreeNode root):
    // Find the grammar rule applied to match the root node
    eruleno = burm_rule(root)

    // Find the children of the current node in the parse tree
    InstrTreeNode children[]
    children = burm_kids(root, eruleno)

    // Traverse the tree bottom-up and collect the attributes of the children
    TreeAttribute child_attrs[]
    for each child in array children
        child_attrs[child] = Reduce(child)

    // Use these attributes to generate code/compute attribute for current node
    return GenCode(root, child_attrs, eruleno)


// This function either generates code for the root node using the attributes of the children
// or/and computes the attribute of the root using the attributes of the children and returns
// the attribute of the root node. If code is generated, the attribute of the root is normally
// the virtual register which holds the value of the root node.
TreeAttribue GenCode(InstrTreeNode root, TreeAttribute[] child_attrs, int eruleno)
```

Figure 3.3: BURG based instruction selection. The actual function for code generation depends upon the grammar rule and is not shown here for brevity.

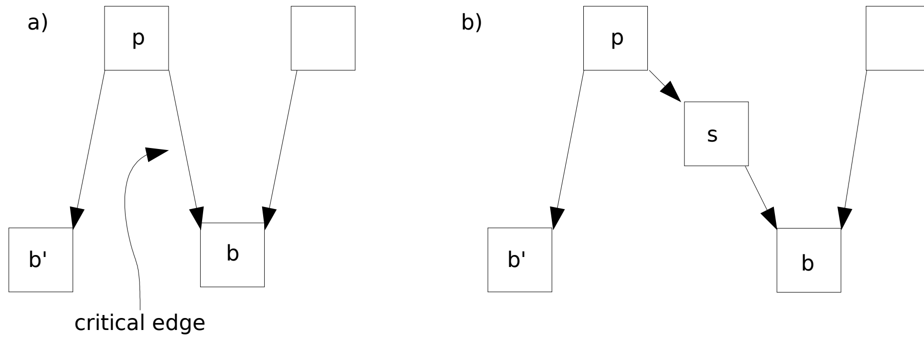

Figure 3.4: Critical edge breaking: (a) a critical edge $p \rightarrow b$, and (b) splitting of critical edge $p \rightarrow b$ by adding a block s between p and b.

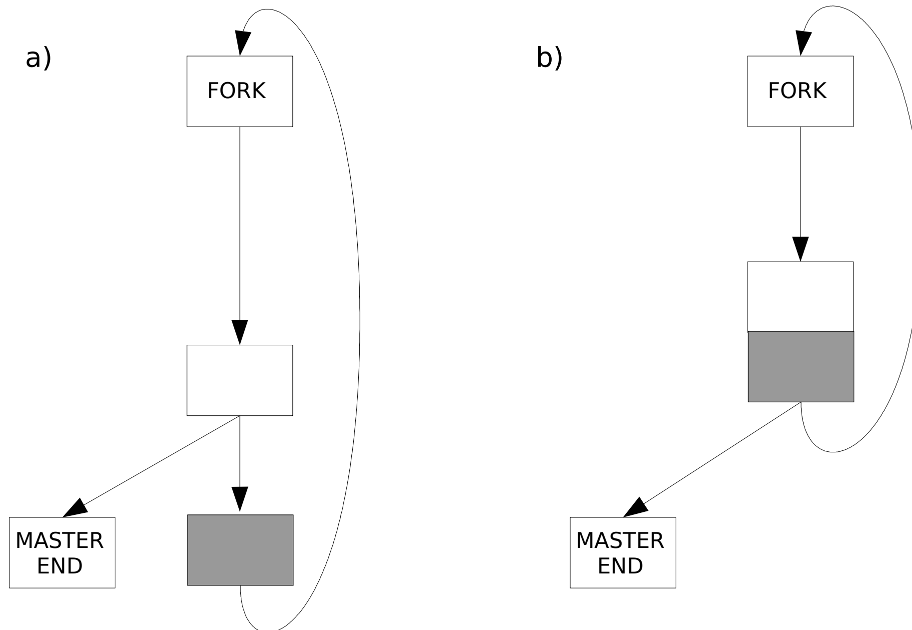

Figure 3.5: Code hoisting to a MASTER_END predecessor: (a) before hoisting, and (b) after hoisting

Chapter 4

Register Allocation

Since the instruction selector assumes an infinite set of virtual registers available for code generation, the optimizer needs to map these virtual registers to the finite set of physical registers available on the Alpha. This register allocation process also has to make sure that certain MSSP-specific requirements are satisfied. The first is that at task boundaries, the task liveouts must be in the register in which the next task expects them to be. The VIR encodes information about the physical registers that need to hold the liveouts by using register copy instruction (from a virtual register to a physical register) just before the VERIFY instruction. The second requirement is that unlike a compiler, the register allocator is not allowed to spill virtual registers. This is because there could be a liveout for the task that resides on the stack and any spilling done by the allocator will change the layout of the stack frame of the optimized trace code and will result in misspeculations because of the mismatch of stack resident liveouts. This chapter explains the *linear scan* register allocation [7] algorithm used by the optimizer and some of the changes made to it to satisfy the requirements imposed by MSSP.

4.1 Linear Scan Register Allocation

Linear scan register allocation is a simple linear time algorithm for global register allocation [7] that uses *live intervals* to represent variable lifetimes. A live interval of a variable is the sections of code in which that variable is live. After live intervals have been computed, the algorithm does a single linear scan over the live intervals and either allocates a register to a live interval or decides to spill it. More details of the linear scan algorithm can be found in [6, 7]. Our basic linear scan register allocation algorithm is the same as the one proposed in [7], augmented with certain heuristics to avoid spilling. In the following sections, we describe some details about our implementation.

4.1.1 Preallocation

As mentioned in previous chapters, the optimizer maintains register allocation hints for the register allocator in the form of register read/write intrinsics in the LLVM IR and register copy instructions in the VIR. Our initial implementation of the linear scan allocator was designed to consider these hints when allocating registers. Specifically, when multiple free registers were available for allocation to the current live interval in the linear scan process, the allocator would look for a register copy to/from a physical register in the uses and definitions of the live interval. If the live interval was

involved in such a register copy and the corresponding physical register was available for allocation, the allocator would allocate that register to the live interval. Although useful, this heuristic was not effective, because many times the register involved in the register move was not free. This was a result of a bad register allocation decision somewhere before the current live range was considered. In effect, the allocator did not exploit the register allocation hints as well as it could.

To prevent this problem, our implementation first performs a *preallocation* step that *enforces* the register allocation hints by preallocating such live intervals to their preferred registers beforehand, instead of during the linear scan process. Whenever the instruction selector creates a register allocation hint in the form of a register copy between a virtual register v and a physical register r, it adds an entry in a preallocation table. Before starting the actual linear scan allocation, the preallocator does a single pass over all the instructions in the VIR, and for each virtual register in the preallocation table, replaces it with the corresponding physical register. This preallocation process helps the actual linear scan algorithm by making certain "bad" register allocation decisions illegal. An example is shown in Figure 4.1.

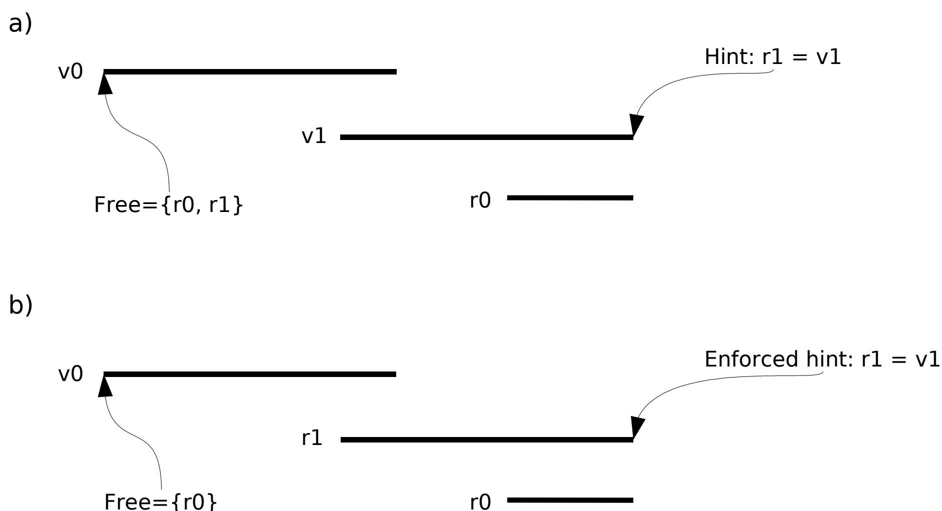

Figure 4.1: Preallocation makes certain bad register allocation decisions illegal: (a) Both $r0$ and $r1$ are free when the live interval $v0$ is considered. If $r1$ is allocated to $v0$, no free registers would be available when live interval $v1$ is considered, and (b) When the allocation hint $r1 = v1$ is enforced by preallocation, only $r0$ is free when the live interval $v0$ is considered for allocation, making the allocation of $r1$ to $v0$ illegal.

Preallocation reduces the number of register allocation decisions that the actual linear scan allocator has to make and thus decreases the probability that register allocation will fail. In general, the more hints the front end can insert, the better are the chances of a successful register allocation. Therefore, when handling function calls, the front end writes all registers before the call and reads back all the registers after the function call. This conservative approach does interfere with the dead code elimination process, but helps in register allocation. As an example, an instruction that writes to an argument register, say `a3`, before a function call may be dead because `a3` is not a livein to the callee. However, in the LLVM IR, when all register are written to before a call, the compiler cannot eliminate that instruction because it sees a single use of the output in a write intrinsic,

which it does not consider dead. This conservative approach helps the optimizer optimize many traces on which the register allocation would fail otherwise. A less conservative approach can be used when the allocator is more careful about the allocation decisions and their effect on future allocations and can aggressively exploit the register allocation hints.

4.1.2 Live Intervals

After preallocation, the allocator constructs live intervals for the virtual and physical registers. Live intervals for physical registers are called *fixed* live intervals. The first step in constructing live intervals is an iterative register liveness analysis [11]. This finds the (physical and virtual) registers that are live at the entry and exit of each basic block in the VIR. Once this information is available, constructing live intervals requires a single pass over the VIR. Each live interval is represented as a set of (sorted) *ranges*, with each range represented by its starting and ending point (both inclusive). An important issue in representing ranges is representing program points. Simple representation like instruction number cannot distinguish between the program point before and after an instruction, or at the end of a block and the start of its successor. We decided to adopt the notion of *slots* [12] to represent program points. Slots can easily represent the differences between program points before and after an instruction or block and make the computation and manipulation of live intervals easier. Our implementation defines two kinds of slots (see Figure 4.2):

- Instruction slots – These slots represent the program points before and after an instruction. Each instruction is defined to have two slots: a *use* slot and a *def* slot. A use slot is the end of the live range for a virtual register which is dead after the instruction. A def slot is start of the live interval of the virtual register defined by the instruction. An instruction slot is represented by a triple <Block Number, Instruction Number, Subslot> where the subslot is 0 for a use slot and 1 for a def slot.

- Basic block slots – These slots represent the program points at the entry and exit of a basic block. Each basic block is defined to have four slots, two at entry and two at exit. The motivation is to allow the merging of live ranges at block boundary and at the same time prevent artificial interference between the live intervals of two variables at block boundary. The four slots are used to start/end the live range of variables as follows:

  - Entry Slot 0 – used to start the live range for a block livein that is also a liveout of the previous block in the linear ordering of the blocks used by linear scan allocation. Not used for entry blocks.
  - Entry Slot 1 – used to start the live range of a block livein that is not a liveout of the previous block and all liveins of entry blocks
  - Exit Slot 0 – used to end the live range of a block liveout that is not a livein to the next block and all liveouts of verify blocks.
  - Exit Slot 1 – used to end the live range of a block liveout that is also a livein to the next block. Not used for verify blocks.

We also define slot equality to enable the merging of live ranges at block boundary. The exit slot-1 of a basic block is defined to be equal to the entry slot-0 of the next block. With this definition, the live interval analysis can merge two live ranges $[a, b]$ and $[c, d]$ if $b = c$.

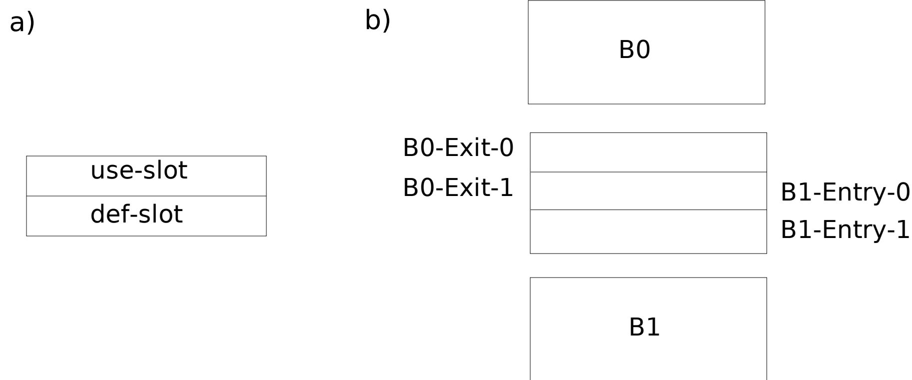

Figure 4.2: Representing program points using slots: (a) instruction slots, and (b) basic block slots. The figure shows that slot B0-Exit-1 and slot B1-Entry-0 are equal.

As mentioned above, entry and verify blocks are treated specially: all liveins to an entry block begin their live range in slot 1 of the entry block and all liveouts of the verify blocks end their live range in slot 0 of the verify block. Figure 4.3 shows the algorithm for constructing live intervals given the register liveness information for each block.

4.1.3 Linear Scan Algorithm

Once the live intervals are constructed, the actual linear scan allocation algorithm is the same as that presented in [7]. As mentioned above, our allocator cannot spill virtual registers. Thus, when the allocator finds that there is no free register available for allocation to the current live interval, it does not spill that register and instead declares a failure in register allocation. In such a case, the optimizer does not optimize the trace, and the simulator continues using the unoptimized trace. However, to prevent such register allocation failures, we have implemented three heuristics that can help the allocator in making better allocation decisions. These heuristics are as follows:

- Register copies between virtual register – If there is a register copy instruction $v1 = v0$ and $v0$ has been allocated a register $r0$, then $v1$ will also be allocated $r0$ if $r0$ is free at the time the live interval for $v1$ is considered.

- Idioms – For some instructions, we can determine the preferred register for their input/output virtual registers and allocate this preferred register if its free. As an example, at the start of a function there can be an Alpha instruction `lda sp, -32(sp)`, which would get translated into the following VIR instruction: `lda vri34, -32(sp)`. Here, the preferred register for `vri34` is `sp`.

- Register constrain – This is a greedy heuristic that helps the allocator choose the correct register when more than one free register is available for allocation. For each physical register, the allocator keeps track of the *unhandled* (i.e., not yet allocated a register) live intervals that overlap with it. It maintains this information in the form of an *interference table*, which is an array of set of live intervals. This interference table is an indicator of how "constrained" the physical register is. Whenever a register is allocated to a live interval, the live interval is removed from the all the interference table entries. When more than one free register

```
// Construct live intervals for the virtual registers in function F
BuildLiveIntervals(VFunc F):
    for each block b ∈ F in linear scan order
        prevLiveouts = liveouts for previous block. φ for the first block or an entry block
        nextLiveins = Liveins for next block. φ for the last block or a verify block.
        blockRange[v] = φ ∀ virtual registers
        for all registers v in liveins(B)
            slot = entry slot 0 if v ∈ prevLiveouts, entry slot 1 otherwise
            blockRange[v] = [slot,slot]

        for all instruction I in block b
            for all input registers in of I
                blockRange[in].end = use slot of I
            for all output registers out of I
                Add blockRange[out] to the live interval for out if its defined
                blockRange[out] = [def slot of I, def slot of I]

        for all registers v in liveouts(B)
            slot = exit slot 1 if v ∈ nextLiveins, exit slot 0 otherwise
            blockRange[v].end = slot
        for all live ranges l in blockRanges
            Add l to the live interval for l.vreg

    Delete any empty intervals
```

Figure 4.3: Algorithm for constructing live intervals given live register information

is available for allocation, the allocator chooses the register that is *most constrained* (the register with maximum interference set size), keeping the less constrained registers free for future allocations.

The order in which these heuristics are applied also affects the success of register allocation. In general, the register copies and idioms are more effective in the entry and verify blocks and register constrain is more effective otherwise. Thus, the allocator assigns a higher priority to copy and idiom heuristics if the block containing the start or the end of the current live interval is an entry or verify block and assigns the register constrain heuristic a higher priority otherwise.

4.2 Rewrite and Code Cleanup

After successful register allocation, the rewrite step is a simple replacement of each virtual register in the VIR with its allocated physical register. This step requires a single pass over the VIR. This rewrite step can render many instructions in the VIR either dead or redundant. Secondly, the register allocated code can be simplified further by eliminating unconditional branches. Thus, after rewriting, the optimizer applies the following simple optimizations:

- Dead instruction elimination – If the output of a non-control-transfer instruction is either `r31` or `f31` and the instruction does not have any side effect (e.g, stores and prefetched) then that instruction is dead and can be eliminated.

- Redundant instruction elimination – After allocation, many register copy instructions are rendered redundant as their source and destination registers are the same. Such instructions are eliminated.
- Unconditional branch elimination – In many cases, after dead and redundant instruction elimination, a block is left with just a single unconditional branch. For example, blocks added to the CFG for splitting critical edges can be rendered like this after allocation. For such blocks, the optimizer eliminates them and modifies all of their predecessors to jump to the single successor of the eliminated block.

After these steps, the optimized trace is in the form of register allocated VIR. The final step in the optimization process is to generate Alpha machine instructions from this VIR and deploy the resulting code in the simulator. The details of this step are described in the next chapter.

Chapter 5

Deployment

Deployment of the optimized code consists of creating Alpha machine instructions from the register allocated VIR, allocating memory for the optimized code, fixing the branch instructions as per the memory allocation, and communicating to the simulator the MSSP specific pieces of information that are required for correct execution of the optimized code. The program orienteering framework [2] ensures that all the branches in the trace code are intra-trace; however there may be function calls from one trace to another and those need to be fixed to resolve to the optimized versions of the trace code. Much of this functionality is provided by the orienteering framework. Thus, deployment of the optimized code from the perspective of the optimizer consists of providing correct information to the orienteer so that it can perform the actual deployment tasks. This consists of converting the trace to be deployed into an IR that is essentially Alpha machine instructions (which we call *OPTIR*) and annotating the OPTIR with the MSSP-specific information present in the VIR. These annotations consist mainly of the original PC for certain basic blocks and instructions and certain flags that need to be preserved throughout the optimization process. This chapter explains some of the prerequisites of the orienteer and the translation of VIR to OPTIR as per these prerequisites.

5.1 Generating Alpha Instructions

One of the basic requirements of the deployment phase is creating OPTIR Alpha instructions from instructions in VIR. This depends upon the type of the VIR instruction for which the code is to be generated:

1. VInst – Since a VInst maintains a pointer to the corresponding OPTIR instruction, code generation for such instructions simply consists of renaming the OPTIR registers as per the register allocation.

2. VInstBranch – For conditional branches, code generation consists of renaming the predicate register of the underlying OPTIR instruction, and informing the orientieer the targets of the branch. The orienteer requires that the fall-through target (i.e., target taken when the branch predicate is false) be specified first and then the taken target. Given this information, the orienteer can add appropriate offset to the branch instruction and reverse the sense of the branch if necessary. For unconditional branches, the orienteer requires that the actual branch instruction should not be present in the OPTIR. Instead, the optimizer just needs to inform

the orienteer that an unconditional branch exists, and the orienteer can add the actual branch instruction.

3. VInstUnimp, VInstUnimpBranch – For these instructions, the original bit pattern is first used to create an OPTIR instruction. Then the input/output registers are renamed as per the register allocation.

4. VInstUnimpFix – Since registers cannot be renamed for these instructions, code generation consists of creating an OPTIR instruction using the original bit pattern of the instruction.

5.2 MSSP-Specific Annotations

Annotating the OPTIR with MSSP-specific information generally consists of straightforward transfer of the annotations from the VIR to the OPTIR. In some cases, the blocks in OPTIR need to be annotated with some flags that must be extracted from the original trace code that was the input to the optimizer. This can be done by adding an LLVM intrinsic for such flags and using them to recreate the necessary flags. However, a better option is to maintain a pointer to the basic block in the input trace code and use it to extract the necessary flags. When this original block is available, it can also be used to find the original PC annotations. Thus, whenever the original PC or flags need to be preserved throughout the optimization process, the front end adds a `mark_addr` intrinsic, with its argument a *pointer* to the block in the input trace. This pointer can then be used to extract original PC, flags, and any other information that the OPTIR needs to be annotated with. Thus, for each basic block in VIR, the optimizer checks if the block has an original block pointer annotated to it. If yes, it transfers the original PC and a subset of flags to the new OPTIR block.

5.3 Fixing Function Calls

During deployment, the function call instructions in the optimized trace need to be changed to point to the appropriate version of the callee. As mentioned before, the optimizer may bail out when optimizing some of the traces. Thus, the optimization process creates new versions of some traces and retains the old versions for some traces. In such a case, function calls in the old traces also need to be updated to point to the optimized versions of the callees, if the callee was optimized. To facilitate this process the optimizer creates a table that maps from the unoptimized trace to the optimized trace. If a trace has not been optimized, the entry in this table maps to the unoptimized trace itself.

The mechanism for fixing up function calls in optimized traces is different from that for unoptimized traces. For optimized traces, the optimizer itself is responsible for modifying the call graph edges to point to the new callees. After all the traces have been processed and the above mentioned map constructed, the optimizer does an additional pass over the optimized traces. During this pass, it finds call graph edges whose source is in an optimized trace and whose destination is a trace that the optimizer was able to optimize. Each such edge is fixed up to point to the optimized version of the callee trace. For unoptimized traces, the orienteer provides a function that, given an unoptimized and optimized callee, can replace all the references to the unoptimized callee with the optimized callee.

After these steps have been performed, the OPTIR is ready to be passed back to the orienteer, which does code layout (allocating memory, assigning addresses, and resolving branches) of the optimized trace, and then writes them to the memory. It also changes the function call instructions in the unoptimized traces and write out these callee-updated traces as well. After this process is complete the optimized code is ready to run in the simulator.

5.4 Results

This section presents some data on the code generated by the optimizer when optimizing traces in a subset of SPEC2000 integer benchmarks. These results were collected using profiled task-liveins. We found that the optimizer was, on an average, able to eliminate 3.2% of the instructions during dead code elimination. As mentioned before, the optimizer bails out and declares a failure when it cannot optimize a trace. We collected statistics about the various causes of bail-out and present them in Table 5.1.

Benchmark	Traces	Unopt Insts	Opt Traces	Opt Insts	% Elim	Forbid	Reg Fail	Tree Fail	No Livein
bzip2	7	420	7	375	10.7	0	0	0	0
crafty	70	4014	62	3943	1.8	0	6	2	0
eon	135	5896	112	5739	2.7	2	10	3	8
gap	1	16	1	16	0.0	0	0	0	0
gcc	160	7399	136	7041	4.8	0	10	0	14
gzip	13	717	13	752	-4.6	0	0	0	0
mcf	1	16	1	16	0.0	0	0	0	0
parser	21	475	18	460	3.2	2	1	0	0
perl	273	11847	202	11347	4.2	5	16	2	48
twolf	31	1541	29	1449	6.0	1	0	1	0
vortex	107	5324	83	4915	7.7	1	1	0	22
vpr	33	2116	32	2070	2.2	1	0	0	0
Avg					3.2				

Table 5.1: Optimization results on a subset of SPEC2000 integer benchmarks: *Traces* is the number of traces collected by the orienteer, *Opt Traces* is the number of traces the optimizer was able to optimize, *Unopt Inst* is the sum of the number of instructions in the optimized traces before optimization and *Opt Inst* is the same after optimization, *% Elim* is the percent of instructions eliminated during optimization, *Forbid* is the number of forbidden traces, *Reg Fail* is the number of traces on which the register allocation failed, *Tree Fail* is the number of traces that generated trees that could not be parsed by the instruction selector, and *No Livein* is the number of traces which were not optimized because no live-in information was available.

As can be seen from the table, the optimizer is able to process a majority of the traces but is not very effective in eliminating dead code. The reason for this is the use of LLVM intrinsic functions to represent instructions that have no direct translation to LLVM IR. LLVM conservatively assumes that these instructions may write to memory and does not eliminate them. In future, LLVM plans to provide a way of specifying the intrinsics that don't write to memory. This will help in

eliminating more dead code and will increase the percent of instructions eliminated.

For some traces, although the optimizer is able to optimize them, the resulting code has problems in deployment. An example is described shortly. For such traces, we choose to exclude them from the optimization by either adding their start address to a list of forbidden traces or setting the liveins for one of the VERIFY blocks in them to an empty set. In both cases, the optimizer bails out and the simulator continues using the original code. We also found that in some traces, the trees created from the LLVM IR cannot be parsed according to the grammar defined by the instruction selector. This is most probably an indicator of a bug or an incomplete grammar. We plan to address this issue in future.

As an example of a trace that the optimizer is able to optimize but fails in deployment, consider the trace in Figure 5.1. Both the indirect branch and the function call have block B as one of their successors. MSSP requires that any successor of an indirect jump or a function call be associated with its original PC. If however, the edge from the JMP to the block B was split as a part of critical edge breaking, the same original PC now maps to two different addresses in the optimized code. Since MSSP maintains a single mapping between the original PC and the distilled PC, the code generated by the optimizer cannot be deployed as is. One solution to this problem would be to modify the LLVM critical edge breaking pass to prevent the breaking of critical edges like $JMP \rightarrow B$ in which the target requires an original PC for resolving indirect branches.

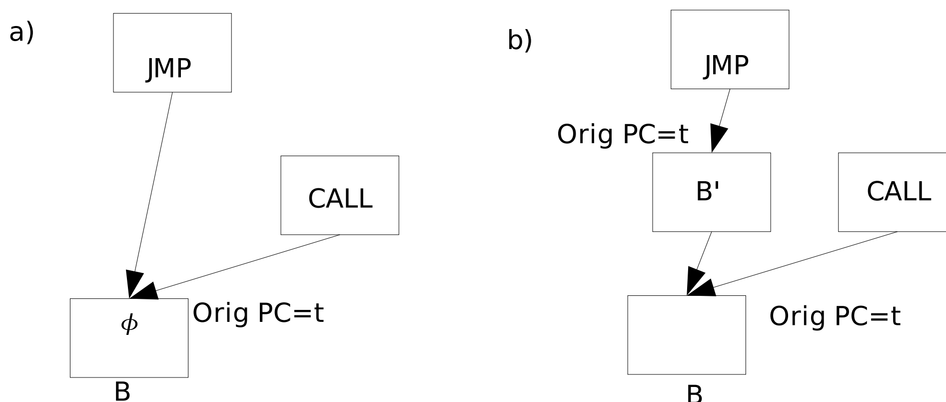

Figure 5.1: Deployment failure when a critical edge whose target is the target of multiple indirect jumps in broken.

We also collected data when the optimizer does not do any optimization (Table 5.2). In such cases, the "optimized" code size can be greater than or less than the original code size. The reason for increase in the code size is the extra register copy instructions introduced during register allocation. Also, when no optimizations are applied, the register pressure is the same as in the original trace code and our simple linear scan allocator fails on more traces than in the case when optimizations are applied. In some case, however, the size of the optimized code is less than that of the unoptimized code. This happens because of the difference in the layout of the unoptimized and optimized traces, which helps in eliminating redundant unconditional branch instructions. The program orienteer can introduce such branches when a trace consisting of two or more nested loops is distilled and the outer loop is distilled before the inner loop. In such cases, the orienteer adds the FORK and VERIFY instructions for the task that starts at the inner loop at the end of the code forming the outer loop (see Figure 5.2). When such a trace is passed through the optimizer, it

Benchmark	Traces	Unopt Insts	Opt Traces	Opt Insts	% Change	Forbid	Reg Fail	Tree Fail	No Livein
bzip2	7	285	6	282	1.05	0	1	0	0
crafty	70	2751	55	2825	-2.69	0	13	4	0
eon	135	5119	104	5254	-2.64	2	16	3	8
gap	1	16	1	16	0.00	0	0	0	0
gcc	160	5189	128	5198	-0.17	0	18	0	14
gzip	13	343	11	358	-4.19	0	2	0	0
mcf	1	16	1	16	0.00	0	0	0	0
parser	21	475	18	468	1.47	2	1	0	0
perl	273	9056	186	9001	0.61	5	23	11	48
twolf	31	1235	28	1231	0.32	1	1	1	0
vortex	107	4376	77	4300	1.80	1	4	3	22
vpr	33	1703	30	1704	-0.59	1	2	0	0
Avg					-0.42				

Table 5.2: Optimization results with all optimizations disabled

chooses a different code layout and is able to "inline" the FORK (Figure 5.2(b)).

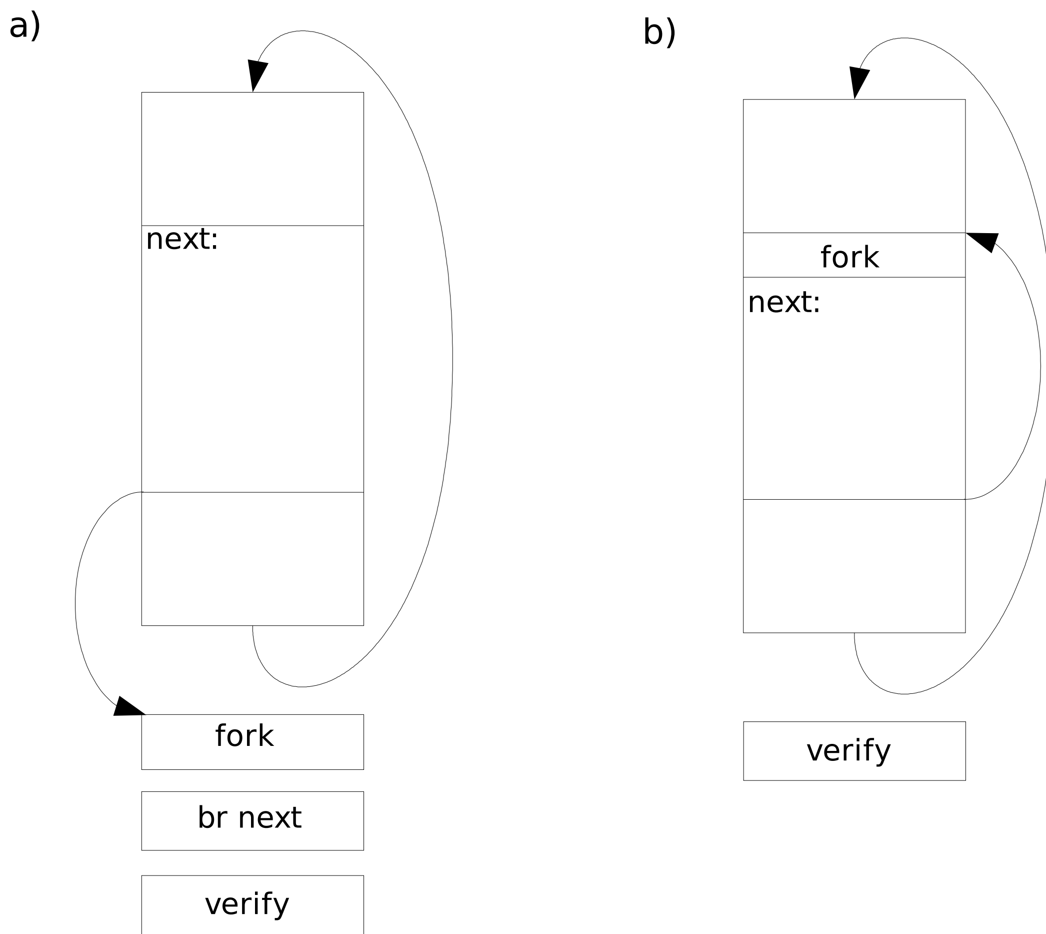

Figure 5.2: Redundant `br` instruction elimination during optimization: (a) a trace with two nested loops, and (b) the optimized trace with “inlined” FORK

Chapter 6

Conclusion

We described the motivation for and the design and implementation of a framework for optimizing code within the context of the MSSP paradigm. Our framework is built on the top of the program orienteer and the LLVM compiler and is designed to address the requirements and restrictions imposed by MSSP. Currently it applies only simple optimizations like dead code elimination and its effectiveness is limited by the use of LLVM intrinsic functions and the simple linear scan allocator.

This framework can be extended and improved in many directions. As mentioned in Chapter 2, the front end can analyze the trace code more rigorously to eliminate `cast` instructions and thus create more compact translations. A more general instruction selector can be constructed by modifying/extending the tree grammar or using an entirely different approach for instruction selection (e.g., translating each LLVM instruction individually into an Alpha instruction). This will enable the application of a number of optimizations provided by the LLVM compiler framework. Our optimizer omits many important steps (like instruction scheduling, code layout, and peephole optimization) in creating high-performance code. In general, this framework can be extended by creating a more sophisticated front-end and back-end and implementing additional approximating transformations during the optimization process.

Another direction for allowing the immediate use of all the LLVM optimizations would be to apply approximations and optimizations on the LLVM IR created by compiling the original sequential code (rather than translating the machine code). This has the advantage that the high level semantic and type information is available in the LLVM IR and that, along with an execution profile, can help in aggressive approximation.

References

[1] C. Zilles and G. Sohi, "Master/Slave Speculative Parallelization," in *Proc. of the 35th International Symposium on Microarchitecture (Micro-35)*, November 2002.

[2] N. Neelakantam, "Program Orienteering," Master's thesis, Univerisity of Illinois at Urbana-Champaign, May 2004.

[3] A. Srivastava and D. W. Wall, "A practical system for intermodule code optimization at link-time," *Journal of Programming Languages*, vol. 1, no. 1, pp. 1–18, December 1992.

[4] R. Muth, S. K. Debray, S. A. Watterson, and K. D. Bosschere, "alto: a link-time optimizer for the Compaq Alpha," *Software - Practice and Experience*, vol. 31, no. 1, pp. 67–101, 2001.

[5] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[6] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895–913, 1999.

[7] H. Mossenbock and M. Pfeiffer, "Linear Scan Register Allocation in the Context of SSA Form and Register Constraints," in *Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag, 2002, pp. 229–246.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[10] C. Fraser, R. Henry, and T. Proebsting, "BURG - fast optimal instruction selection and tree parsing," *SIGPLAN Notices*, vol. 27, no. 4, pp. 68–76, April 1992.

[11] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[12] A. Evlogimenos, "Improvements to Linear Scan register allocation," 2004, CS426 Project Report. [Online]. Available: `llvm.cs.uiuc.edu/ProjectsWithLLVM/`