University of Kentucky
UKnowledge

University of Kentucky Master's Theses
Graduate School

2010

GENETIC ALGORITHM CONTROLLED COMMON SUBEXPRESSION ELIMINATION FOR SPILL-FREE REGISTER ALLOCATION

Shashi Deepa Arcot
University of Kentucky, shashiarcot@gmail.com

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Arcot, Shashi Deepa, "GENETIC ALGORITHM CONTROLLED COMMON SUBEXPRESSION ELIMINATION FOR SPILL-FREE REGISTER ALLOCATION" (2010). *University of Kentucky Master's Theses*. 641.
https://uknowledge.uky.edu/gradschool_theses/641

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

GENETIC ALGORITHM CONTROLLED COMMON SUBEXPRESSION ELIMINATION FOR SPILL-FREE REGISTER ALLOCATION

As code complexity increases, *maxlive* increases. This is especially true in the case of the *Kentucky If-Then-Else* architecture proposed for Nanocontrollers. To achieve low circuit complexity, computations are decomposed to bit-level operations, thus generating large blocks of code with complex dependence structures. Additionally, the Nanocontroller architecture allows for only a small number of single bit registers and no extra memory.

The assumption of an infinite number of registers made during code generation becomes a huge problem during register allocation because the small number of registers and no additional memory. The large basic blocks mean that *maxlive* almost always exceeds the number of registers and the traditional methods of register allocation such as instruction re-ordering and register spill/reload cannot be applied trivially. This thesis deals with finding a solution to reduce *maxlive* for successful register allocation using Genetic Algorithms.

KEYWORDS: Nanocontrollers, Register Allocation, Sethi-Ullman Numbering, Genetic Algorithms, Maxlive Reduction

Shashi Deepa Arcot

12/08/2009

GENETIC ALGORITHM CONTROLLED COMMON SUBEXPRESSION ELIMINATION FOR SPILL-FREE REGISTER ALLOCATION

By

Shashi Deepa Arcot

Dr. Henry G. Dietz
Director of Thesis

Dr. Stephen D. Gedney
Director of Graduate Studies

12/08/2009

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regards to the rights of the authors. Bibliographical references may be noted, but quotations and summaries of parts may be published only with the permission of the author and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name | Date

THESIS

Shashi Deepa Arcot

The Graduate School

University of Kentucky

2010

GENETIC ALGORITHM CONTROLLED COMMON SUBEXPRESSION ELIMINATION FOR SPILL-FREE REGISTER ALLOCATION

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering in the College of Engineering at the University of Kentucky

By

Shashi Deepa Arcot

Lexington, Kentucky

Director: Dr. Henry G. Dietz, Professor of Electrical and Computer Engineering

Lexington, Kentucky

2010

Copyright© Shashi Deepa Arcot 2010

To Amma and Dad

ACKNOWLEDGEMENTS

I wish to acknowledge my thesis advisor, Dr. Henry Dietz, whose guidance and patience has made it possible for me to successfully complete my thesis. Not only is he an excellent researcher and a teacher, but above everything else he is simply a great human being.

I also wish to thank my committee members Dr. Ingrid St.Omer and Dr. Robert Adams for being on my thesis committee.

And lastly, I wish to thank my family for helping me get through tough times and for always being there.

Table of Contents

Acknowledgements iii
List of Tables v
List of Figures vi
1 Introduction 1
1.1 Motivation and Preliminary Work 3
1.1.1 Programming Language: BitC 3
1.1.2 Compiler Technology 4
1.1.3 Kentucky If-Then-Else (KITE) Architecture 6
2 Background 10
2.1 Register Allocation Methods 10
2.1.1 Graph Coloring and its extensions 10
2.1.2 Heuristics for Directed Acyclic Graphs 11
2.1.3 Generalizations of the Sethi-Ullman Numbering algorithm 12
2.2 Initial Register Allocation Attempts for KITE Architecture 13
2.2.1 Register Allocation without Reordering 13
2.2.2 Genetic Algorithm Based Reordering 14
3 Max Live Reduction 16
3.1 Sethi-Ullman Numbering 16
3.2 Genetic Algorithms 19
3.3 Sethi-Ullman Numbering Extension 21
3.3.1 Tree Generation 22
3.3.2 Extended Sethi-Ullman Numbering 24
3.3.3 Register Allocation using SUN 26
3.3.3.1 Algorithm 1 26
3.4 Extended Sethi - Ullman Numbering with Genetic Algorithm 28
3.4.1 Extended SUN using Genetic Algorithm 28
3.4.1.1 Algorithm 2 29
3.4.2 Extended SUN: Tree Re-Ordering 32
3.4.2.1 Algorithm 3 33
3.5 Generalization of Extended SUN for of n-tuples 34
3.5.1 Labeling Rules for an n-ary tree 35
3.5.2 Register Allocation Algorithm for an n-ary tree 36
3.5.2.1 Algorithm 4 36
4 Results 38
4.1 Results 38
4.1.1 Trial 1 39
4.1.2 Trial 2 41
4.1.3 Trial 3 44
4.2 Effect of increasing available registers on CSEs and SITES 46
5 Conclusions 50
5.1 Application to other architectures 51
5.2 Future Work 51
References 52
Vita 53

List of Tables

Table 1.1 Logic Operations and ite structure 6
Table 3.1 SUN Labeling Rules 18
Table 3.2 Extended SUN Labeling Rules...25
Table 3.3 Labeling rules for a generic tree...35

List of Figures

Figure 1.1 KITE Architecture 7
Figure 3.1 SUN Example 19
Figure 3.2 *ite* to *site* conversion 23
Figure 3.3 DAG Representation of sites 23
Figure 3.4 Tree representation of sites 24
Figure 3.5 Tree labeling using extended SUN 26
Figure 4.1 Trial 1 - maxlive 40
Figure 4.2 Trial 1 - *site*s..41
Figure 4.3 Trial 2 - maxlive 42
Figure 4.4 Trial 2 -*site*s..43
Figure 4.5 Trial 3 - maxlive 45
Figure 4.6 Trial 3 - *site*s..45
Figure 4.7 Effect on register limit on CSEs - SUN-GA 47
Figure 4.8 Effect of register limit on CSEs – SUN-GA-TREEREORDER 48
Figure 4.9 Effect of register limit on sites - SUN-GA 49
Figure 4.10 Effect of register limit on sites - SUN-GA-TREEREORDER.........................49

1 Introduction

Register Allocation is a crucial step in the compilation process. In conventional computer architecture, the total number of registers is limited whereas data memory is relatively large. Typical instruction sets and compilation techniques commonly produce basic blocks containing a small number of instructions – generally, fewer than 20 – so the number of registers needed to hold all the values referenced within a block is relatively small. Where the number of available registers is insufficient, an exhaustive search for an instruction reordering may be applied to reduce the number of registers needed. When that search is impractical or fails, register spill/reload can be used.

Larger basic blocks usually require more registers and also make instruction scheduling by exhaustive search computationally infeasible. Larger blocks can result from specific compiler optimizations, such as loop unrolling, or it can be a natural consequence of having a very simple instruction set. For example, without a spill-free register allocation, unrolling might actually yield a slowdown rather than a speedup. If the larger block size was caused by a simpler instruction set, there is also a possibility that the instruction set was not the only thing simplified: the data memory may be of very limited size or completely absent. The absence of an external data memory is one of the main features of a simple architecture called the Kentucky If-Then-Else (or *KITE*) architecture that was proposed to reduce hardware complexity [Die03]. The KITE architecture has a limited number of single bit registers and no external data memory. The unique hardware architecture and the specialized compiler that is required for generating executable code result in large and complex basic blocks. With no external data memory, the process of register spill/reload is not an option. Failure to find a spill-free allocation results in user programs not being executed and thus makes register allocation the most critical part of the KITE architecture.

Previous work on the KITE architecture and the associated compilation techniques reduced the hardware complexity and successfully generated code –

but maxlive was often orders of magnitude larger than the register file could support. No existing technique was able to directly solve the problem of register allocation with a large maxlive that resulted from code generation. This research is aimed at finding a solution to the problem of register allocation for very complex instruction blocks with extreme register pressure and a lack of external data memory. Specifically, the current thesis has focused on the fundamental problem of trying to reduce maxlive enough to fit the KITE architecture's very modest register file.

Our approach is grounded in earlier work. A popular technique developed in 1970 called Sethi-Ullman Numbering (SUN) is used to find the minimum number of registers required and the instruction order for a tree. Modern compilers apply Common Subexpression Elimination to generate code in the form of a Directed Acyclic Graph (DAG), thus minimizing instruction count with the side effect of often increasing maxlive. In this research we converted compiler generated DAGs to trees by replicating the common subexpressions (CSE) whenever the common subexpressions are referenced, thus reducing maxlive. We extended SUN technique and applied it to the trees generated from the DAGs to calculate the minimum number of registers and instruction count required to evaluate a tree. The conversion from DAGs to trees to reduce maxlive resulted in many registers of the register file being unused and also increased the instruction count. We then selectively enabled re-factoring of common subexpressions to minimize the instruction count while keeping maxlive less than or equal to the number of available registers. A Genetic Algorithm (GA) is used to find a solution that contains a set of enabled common subexpressions that would keep the maxlive within the register limit as well as reduce the instruction count. A genetic algorithm is a search technique used to find an acceptable solution to problems that have complex and large search space. The sheer number of common subexpressions generated for the complex basic blocks of the KITE architecture makes the problem of finding the best possible set of the common subexpression to be enabled non-trivial and therefore a GA is

best suited to find a set of enabled common subexpressions. In addition, another Genetic Algorithm is applied to reorder the trees to find a tree execution order that may result in a better maxlive and instruction count.

1.1 Motivation and Preliminary Work

Nanotechnology makes it possible to assemble nanostructures into a wide range of devices, such as chemical/biological sensors, and also to place millions of these devices on a single chip. Intelligent control of these devices requires independent programmability of a controller for each device. Conventional micro-controllers are not small enough to be paired on-chip with each of the devices. The obvious alternative, routing the signals from these devices to off-chip controllers, is often impractical for reasons of speed, signal quality, or wiring complexity. A nanocontroller architecture proposed as a solution to overcome the existing micro scale hardware limitations is called KITE: *Kentucky If-Then-Else* architecture [Die03]. This approach uses a new compiler technology to dramatically simplify the target architecture, yielding circuit complexity on the order of 100 transistors per nanocontroller. Such a simple architecture requires special code that operates at single bit levels. BitC, a subset of C programming language, was developed for real-time, intelligent device control. BitC uses advanced compiler technologies such as Meta State Conversion [DiK93] and Common Subexpression Induction [Die92] to generate large blocks of code.

This section describes the KITE Nanocontroller architecture and the compiler technology associated with it. The user level programming interface is first described followed by the compiler technology and the underlying hardware model.

1.1.1 Programming Language: BitC

BitC, a sequential programming language and a subset of C programming language, is developed for KITE Nanocontroller architecture. BitC allows explicit declaration and/or typecasting of bit precisions. For example,

int: 3, a;

declares *a* as a 3-bit signed integer value. All the C operators are supported with the standard precedence. Additional operators like minimum, maximum and population count are also provided. The bit level manipulation of word level objects, which is the main feature of the KITE Nanocontroller architecture, is hidden from the programmer. Input/Output operations can be done using application-specific reserved registers. Reservation of such special registers is done before allocation of any ordinary variables. Inter-processor communication is also implemented using reserved registers.

1.1.2 Compiler Technology

The compilation of BitC for KITE is a complicated process that involves a large number of transformations. The first step is transformation of word-level operations into simple single bit operations. The bit-slice operations are optimized and simplified using a variety of techniques such as conventional compiler optimization and hardware logic minimization. The optimized bit slice versions of all the programs are merged logically to produce guarded SIMD (Single Instruction, Multiple Data) code using Meta State Conversion (MSC). After Common Subexpression Induction (CSI), the instructions are ordered and allocated to registers as the final step.

All the single bit operations are implemented as an if-then-else tuple or *ite*, a 1-of-2 multiplexor function similar to that of a hardware logic minimization technique. As an example, consider the following BitC code:

unsigned int: 2 a, b, c;

c = a + b;

The BitC code shows a simple addition operation of two 2-bit operands. A bit-level 2's complement addition of operands *a* and *b* generates a 2-bit result *c*, the two bits computed as:

$$c_0 \leftarrow (a_0 \text{ xor } b_0)$$

$$c_1 \leftarrow ((a_1 \text{ xor } b_1) \text{ xor } (a_0 \text{ and } b_0))$$

This bit-level representation uses 2 operators: XOR and AND. The XOR and AND operations can be represented as if-then-else tuples, referred to as *ite*s in the BitC and KITE architecture. An *ite* is shown using C's ternary operator syntax (x_1 ? x_2 : x_3). Table 1.1 shows the *ite* equivalents for most commonly used logic operations.

The 2-bit addition example can be written using *ite*s as:

$$c_0 \leftarrow (a_0 \, ? \, (b_0 \, ? \, 0 : 1) : b_0)$$
$$c_0 \leftarrow ((a_1 \, ? \, (b_1 \, ? \, 0 : 1) : b_1)$$

$$? \, ((a_0 \, ? \, b_0 : 0) \, ? \, 0 : 1) : (a_0 \, ? \, b_0 : 0))$$

The 2-operand, 2-bit addition example generated 8 *ite*s. Common *ite*s such as a_0 ? b_0 : 0 are equivalent to common subexpressions and are reduced by Common Subexpression Elimination (CSE) process. The total number of instructions is further reduced by combining explicit store (into variable) instructions and *ite*s, thus creating a new structure called store-if then-else tuple (*site*). The conversion from *ite*s to *site*s is an important step because s*ites* can be converted from DAGs to trees, and a tree structure is required to extend SUN and apply it for the register allocation of the ternary operators generated by the BitC compiler for the KITE architecture.

As the complexity of the operations and the operand sizes increased, the number of *site*s (instructions) generated per block also increased. With the limited hardware that is proposed by the KITE architecture, the large block sizes and block complexities pose a significant problem during the register allocation phase. The focus of this research is to solve the problem of register allocation for the complex code blocks generated by the bit-level operations. The BitC compilation can be summarized as:

1. Word to bit level transformation and logic minimization
2. Generation of *ites* and *sites*
3. Register Allocation and Code Scheduling

Steps 1 and 2 have been implemented earlier [Die03] and Step 3 is the topic of current discussion.

Table 1.1 Logic Operations and ite structure

Logic Operation	Equivalent ITE structure
(x AND y)	**(x ? y : 0)**
(x OR y)	**(x ? 1: 0)**
(NOT x)	**(x: 0 : 1)**
(x XOR y)	**(x ? (y : 0 : 1) : y)**
((NOT x) ? y : z)	**(x: ? z : y)**

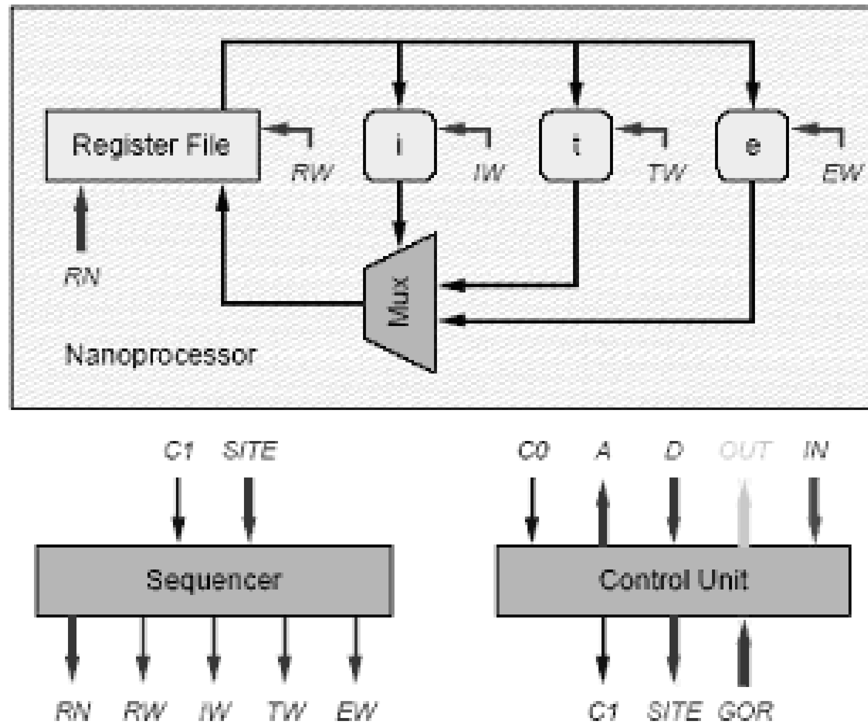
1.1.3 Kentucky If-Then-Else (KITE) Architecture

The main goal of the KITE project is to reduce the hardware complexity and achieve MIMD (Multiple Instruction, Multiple Data) programmability. This can be achieved by the specialized compiler developed for KITE architecture that is described in section 1.1.2. The target architecture is a very simple hardware model. There are 3 main components to the hardware: Control Unit, Sequencer and a Nanoprocessor. A detailed description of the hardware model can be found in [Die03]

The Control Unit: The control unit in KITE architecture controls the program memory interface and not the processors as in traditional SIMD architectures. Implementation of MIMD programs across all the processors is done using MSC which results in a very large meta-state automaton consisting of large basic blocks of *ite*s that end in *k*-way branches rather than binary branches. State transitions from the current state to the next meta state are decided by a *Global OR (GOR)* of votes from all the processors. The large meta state programs generated by the compiler are loaded into off-chip memory that is interfaced by conventional address (*A*) and data (*D*) buses. The controller would perform decompression, branch prediction, and instruction cache management treating each basic block as a single unit. This allows the control unit to pre-fetch code

chunks using a relatively slow clock (*C0*) determined by the external memory system, while internally broadcasting partially decoded instructions (*site*s) from cache at a significantly faster rate (*C1*).

Figure 1.1 KITE Architecture



The Sequencers: The purpose of the sequencers is to make a slow broadcast of *site*s to the nanoprocessor. Thus, there would be many sequencers, each hosting a moderate number of nanoprocessors. The *site* representation of an instruction actually is a compact form that generates four consecutive clock cycles worth of control information for the nanoprocessors. Thus, the input clock (*C1*) to a sequencer can be as much as four times slower than the nanoprocessor clock. More precisely, a particular sequencer's control line outputs imply a "clock" for the nanoprocessors, but nanoprocessors are only loosely synchronized across sequencers. Incorporating additional nanoprocessors and sequencers could also provide a means for fault tolerance by disabling the sequencer above each faulty component.

The Nanoprocessor: The nanoprocessor consists of 1-bit registers, a single register-number decoder and a 1-of-2 multiplexor. The operation of a multiplexor is analogous to the software concept of an if-then-else; if the value in **i** is true, return **t**, else return **e**. The value returned by the multiplexor can be stored in any register selected. The *site* representation of an instruction is literally four register numbers: the register to store into, one to load **i**, one to load **t**, and one to load **e**.

The sequencer simply converts that into a four-cycle sequence using *RN* to specify the register number for the decoder and using the other lines to latch a value into the corresponding register. Registers 0 and 1 are not registers, but respectively generate the read-only constants 0 and 1. Similarly, for each application, a KITE Nanocontroller will require specific network connections and local input and output registers; these are addressed like registers starting with register number 2. The minimum number of bits in a KITE register file is thus the sum of 2 constant registers, the number of additional registers needed for network and local input and output, the ceiling of log2 of the total number of nanoprocessors in the system (for the control state), and the maximum number of ordinary data bits required in any nanoprocessor. Given the above, a slightly smarter sequencer could be used to opportunistically reduce the total number of clock cycles required from 4 per *site* to as few as one — the result store cycle. For example, if the same register number is used to load both **i** and **t**, the loading of both can be accomplished in a single clock cycle. Further, if the current *site* duplicates fields from the previous one, and those fields do not correspond to network or local input or output accesses, the sequencer can skip loading of any of **i**, **t**, or **e**. Such a sequencer would need to buffer incoming *site*s to compensate for variability in the rate at which it processes *site*s, but execution time would still be predictable because the optimization opportunities depend only on the *site* sequence coming from the control unit.

The proposed KITE Nanocontroller architecture consists of a 64-bit register file. Hence the first 64 *ite* index values represent the registers. ITE0 and

ITE1 are used to represent constants 0 and 1 respectively at registers 0 and 1. Network registers, input/output registers and the user-defines variables are represented beginning from *ite* index 2. The *ite* operations are represented from index 64.

2 Background

This chapter describes the more traditional approaches to register allocation and the early attempts at register allocation for KITE nanocontroller architecture. With fewer than 64 single-bit registers available to hold the variables and temporary intermediate values in addition to the complex basic blocks with thousands of *site*s (instructions) per basic block, the traditional register allocation methods proved to be inadequate.

2.1 Register Allocation Methods

The problem of register allocation involves finding an optimal assignment of available registers within the hardware and/or software constraints. Numerous approaches have been proposed to solve this problem. In the following sections a few of the popular register allocation methods that were explored to solve the register allocation problem of KITE architecture are explained.

2.1.1 Graph Coloring and its extensions

Register allocation via graph coloring was implemented by Chaitin *et al* and is still a popular approach to register allocation. Chaitin's register allocation algorithm consists of live range construction, interference graph construction, coalescing, spill cost estimation and graph coloring.

The live range of each virtual register is first determined followed by generating an interference graph. The interference graph consists of one node for each live range created. The graph also consists of arcs representing interferences between two different live ranges. Once a stable interference graph has been generated, a spill weight is calculated for each live range. Chaitin's register allocator assigns a weight to each live range that represents the cost of spilling it, which is the cost of executing the loads and stores that must be inserted if the live range were to be spilled. When it is necessary to spill a node if a register is unavailable, these estimates are used to select the live range to be spilled.

The actual coloring process in Chaitin's allocator is relatively simple. If there exists a node v, such that the deg(v) < n, assuming a target processor with n registers, then the node and all of its interferences are removed from the graph and placed on a stack. If there are no nodes with deg(v) < n, then a node is chosen to be spilled.

Optimal graph coloring is not simple and several enhancements have been proposed to improve the allocation results. The main goal of most proposed heuristics for graph coloring based register allocation is to minimize the number of spilled nodes and the resulting spill code. Chaitin’s node-removal algorithm [Cha82] is one such example that attempts to minimize the spill instructions inserted and maximize the number of interferences removed from the graph to select the spill nodes. This method of graph coloring does not necessarily produce the best allocation in all situations. The actual number of spill/reload events depends on the precise reference sequence, not just (potential) overlap of lifetimes. Thus, costs are approximate. Many such variations to the graph coloring method exist that use Genetic Algorithms [FIL97].

2.1.2 Heuristics for Directed Acyclic Graphs

Typically, basic blocks that are generated can be represented by Directed Acyclic Graphs (DAGs). If the DAG is a tree, then a well-known algorithm by Ravi Sethi and Jeffrey Ullman (described in detail in Chapter 3) is used to generate an optimal evaluation in linear time. The problem of generating an optimal evaluation for a given DAG is NP-complete. To generate a good evaluation order for a DAG that is not a tree, this heuristic uses a mix of several simple evaluation strategies that also include a randomized evaluation selection. These simple evaluation strategies are applied concurrently and the best evaluation generated is selected. The idea behind this approach is that there exists no uniform heuristic that generates good evaluations for every possible DAG, but most of the DAGs encountered in real programs belong to one of a few simple classes. For each of these classes, there exists a simple algorithm that generates good, often optimal evaluations. By running these simple algorithms "in parallel" and

choosing the best result, this method aims to obtain a heuristic that copes with most of the DAGs encountered in real programs.

2.1.3 Generalizations of the Sethi-Ullman Numbering algorithm

The Sethi-Ullman Numbering algorithm [SeU70] determines an order of computation of the nodes of the tree that uses the fewest possible registers, subject to following assumptions:

1. The properties of the arithmetic operators are not considered; that is, no arithmetic identities are used.
2. All registers are equivalent; there are no operations that can produce results only in certain registers.
3. The tree is a binary tree: each internal node has exactly two children.
4. The value of each node will fit in one register.

The four conditions listed above can be overly restrictive in real compilers. Several extensions to Sethi-Ullman Numbering have been suggested such as *Generalizations of the Sethi-Ullman algorithm for register allocation* [ApS86]. In the paper two generalizations are proposed. The first generalization is to remove the restriction on the degree of the nodes. The second generalization is to remove the restriction on the size of the computed result.

Each subtree is evaluated first. The number of registers required to compute the parent is the larger of the results of the first subtree evaluation and the specified sum of the results of the rest of the subtrees. The result for the tree is the minimum of results over all permutations of the tree orders since the trees are not just binary. The paper claims that an optimal permutation turns out to be no more difficult than sorting k numbers. The problem of finding an optimal permutation is not trivial as size of the basic blocks increases. This method also relies on register spill-reload as did the original Sethi-Ullman numbering.

Most of the existing register allocation methods assume spill-reload and provide solutions for reducing the cost of spill-reload. Spill-reload solutions are

not applicable to KITE architecture because of the absence of external memory. Others assume simple basic blocks and hence propose algorithms where it is relatively easier to find a schedule. With basic blocks containing instruction count in the order of thousands, such algorithms could not be used for KITE architecture.

2.2 Initial Register Allocation Attempts for KITE Architecture

In the early stages, a few simple approaches were used to test for successful register allocation and get an idea of the size and scope of register allocation for KITE architecture. A simple straight-forward register allocation method and a Genetic Algorithm based reordering of the *ite*s were used to get an estimate of the complexity of the problem.

2.2.1 Register Allocation without Reordering

The very first attempt made at register allocation for BitC compiler-generated code was a simple optimal basic block register allocation scheme without reordering. It used no special techniques and was principally used to get an estimate of the size of the problem. No instruction scheduling was done for the *ite*s that were generated; the instructions were scheduled in the order in which the compiler naturally generated them.

The algorithm was a simple 2 pass scan. In the first scan, a schedule was built for the *ite*s and in the second scan register allocation was performed. This method proved to be impractical, mainly because of the sheer size of a single block of instructions and the extreme register pressure associated with the KITE architecture. For example, a simple 2-operand, 8-bit multiplication operation generated a basic block consisting of about 3000 instructions that were DAGs and a large maxlive. While the basis of KITE architecture and the BitC compiler is the reduction of all operations to single bit level, which resulted in the large basic blocks, no amount of further optimization of code would reduce the basic blocks to sizes where simple register allocation methods would be applicable. Ruling out such a tremendous reduction of block sizes pointed to the other obvious problem

– maxlive. The next step was to reorder the instructions so that the maxlive could be reduced. While a number of valid schedules can be found, a simple reordering to generate a valid schedule resulted in the maxlive of about 700, which is still a very large number. The large block sizes made the process of finding the best schedule from a very large number of search space very difficult.

2.2.2 Genetic Algorithm Based Reordering

To solve the problem of selecting a best schedule from a large set of permutations, a Genetic Algorithm based *ite* reordering was used. A Genetic Algorithm consists of generating random solutions for a given problem and evaluating each solution to select the best. Genetic Algorithms are explained in detail in section 3.2.

To eliminate the cost of generating a random schedule and checking for its validity, only valid schedules were generated. The genome for such a GA is an integer priority that was assigned to each instruction. A schedule was generated by inserting a schedulable instruction with the highest priority at each instruction slot. A population of valid permuted schedules was created. The metric for each population member is evaluated. A metric represents a measure of the validity of each schedule. For example, the metric may be a combination of whether the schedule can be successfully allocated and how far off it is from being successfully allocated. Or, the metric could represent maxlive – a larger maxlive representing a poorer schedule. After the metric evaluation, a number of methods may be used, such as a sorted order or tournament selection etc., to select members for mutation and crossover operations (mutation and cross over operations are explained in section 3.2). These operations reassign the priorities or mix the priorities of the parents to generate new population members. The schedule with the best metric at the end of the Genetic Algorithm was the chosen solution.

A Genetic Algorithm based instruction reordering did not actually reduce the number of instructions. Although, such a reordering found a schedule with

lower maxlive, the maxlive was still around 250. Such a number is still too high for the KITE architecture. Additionally, there is a possibility that the Genetic Algorithm may not converge to a valid solution.

3 Max Live Reduction

Chapter 1 described the hardware profile for which this research is specifically targeted. As mentioned earlier, the total number of registers allowed in the architecture is limited to 64. Registers 0 and 1 are hard-coded to represent ITE0 and ITE1 and hence cannot be used for data storage or register allocation. Of the remaining, some registers are used for input/output operations and for holding variables. Some additional registers are also used to hold state information. Using the registers for various purposes leaves fewer than 64 registers for temporaries during register allocation. As mentioned in Chapter 1, KITE architecture and its compiler technology generate very large basic blocks. With no external data memory, implementing instruction re-ordering or register spill-reload is not practical or trivial. In this chapter we look at the previously mentioned Sethi-Ullman numbering and the modification that this research applied for successful register allocation.

3.1 Sethi-Ullman Numbering

Chapter 2 described the initial attempts that were made for register allocation of *ite*s. These attempts made it clear that the biggest obstacle that was the large maxlive and any solution for successful register allocation should aim to reduce maxlive. The Genetic Algorithm based instruction reordering described in section 2.2.2 also established the fact that mere instruction reordering does not reduce the maxlive down to a number where the available registers can be used without the need for register spilling. Sethi-Ullman Numbering provides an algorithm to find the minimum number of registers required for evaluation of binary trees. This section explains the Sethi-Ullman numbering in detail.

Ravi Sethi and Jeffrey Ullman developed an algorithm called the Sethi-Ullman Numbering (SUN) that can be used to generate optimal code for arithmetic expressions [SeU70] in 1970. The assumptions made by SUN are straight forward and can be met even today by most computer designs. The algorithm assumes that there are $N \geq 1$ general-purpose registers available, each

of which may be interchangeably used as source or destination in an operation. The algorithm for register allocation deals with single arithmetic expression involving binary operations. Each arithmetic operation can be expressed as a binary tree that links each binary operation to two operations that provide its operand values. Leaf nodes in a tree represent initial values and constants.

The SUN algorithm proceeds in distinct phases. In the initial phase, each node is labeled with a number, according to a set of rules (described below). The label of each node represents the minimal number of registers required to evaluate the subtree rooted at that node without requiring any stores (i.e., without register spill/reload). The labels are then used to order node evaluation and register allocation.

A bottom-up walk of the tree is done to assign each node η with a label $L(\eta)$. Table 3.1 shows the rules that are used to determine the label for each node of a binary tree. Rule 1 implies an additional assumption that the binary instructions have a register-memory model in which the right descendant can be accessed directly from memory, provided that the left descendant has been loaded into a register. In other words, an instruction can be of the form reg <- reg + mem, absorbing the fetch of the right operand into the parent instruction. However, rule 1 can be adjusted to accommodate architectures without register-memory instructions by simply assigning any leaf node with a label of $L(\eta) = 1$. Rule 2 reflects use of register- register operations for nodes that are not leaves.

After the tree is generated and the nodes are labeled, the algorithm proceeds to the second phase of evaluating the tree for register allocation. The evaluation is done as a recursive walk, starting at the root node and then selecting an evaluation order for the descendants of each node such that the one with higher label is executed first. The actual register allocation and output of the instruction schedule is done as the recursion unwinds from the leaf nodes of the tree. Since the label on each node is the maximum of live values in the subtree

rooted at that node, provided that the label does not exceed the number of registers available in the architecture, it is trivial to assign a register to each node. If the label on any node exceeds the number of registers in the architecture, SUN provides a simple solution in which values can be selected to be spilled from registers to memory and reloaded when necessary.

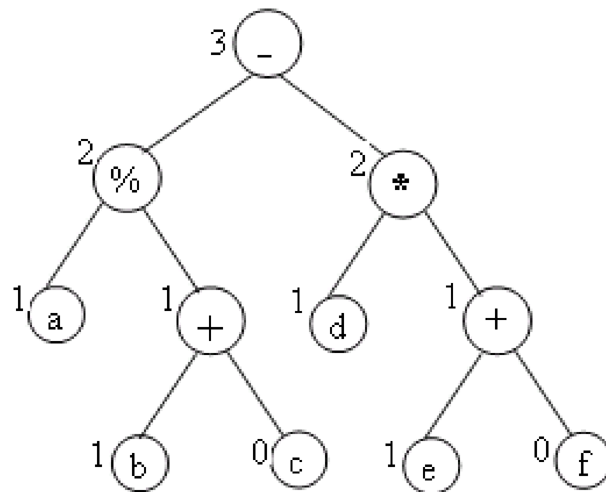
Table 3.1 SUN Labeling Rules

Node Condition	Rule
1. η is a leaf	1. If η is left descendant, $L(\eta) = 1$. 2. If η is a right descendant, $L(\eta) = 0$;
2. η is a not a leaf	If η has descendants with labels l_1 and l_2, 1. If $l_1 \neq l_2$, $L(\eta) = \max(l_1, l_2)$; 2. If $l_1 = l_2$, $L(\eta) = l_1 + 1$;

The entire procedure visits each node at most a constant number of times resulting in an O(n) complexity for instruction scheduling and register allocation for n operations. Given the assumptions that were made for SUN, the algorithm results in an optimal solution for the instruction count and the register count needed for the evaluation of an arithmetic expression. Figure 3.1 shows an example binary tree for the arithmetic expression:

$$(a \% (b + c)) - (d * (e + f)).$$

Figure 3.1 SUN Example



Unfortunately, the assumption that registers can be spilled does not hold true for register allocation for KITE architecture because of the absence of external data memory. In addition, SUN cannot be directly applied to the basic block generated by BitC compilation because SUN assumes that the trees to be evaluated are binary trees where as BitC generated ternary operations in the form of DAGs. So, even the node labeling rules, as described in Table 3.1 cannot be applied to BitC generated code without some extensions.

3.2 Genetic Algorithms

Genetic search algorithms follow the Darwinian principle of *Natural* Selection to evaluate potentially good designs using computer simulation so as to find a good solution to solve complex engineering problems. A set of solutions to a problem constitutes the population. The term genotype is used to refer to the internal representation of the relevant characteristics of the population. The term phenotype refers to the external characteristics of each individual of the population. A fitness or metric is calculated for each individual. The metric usually refers to how close an individual of the population is to the desired final solution. It is used for assigning a rank to all the individuals of the population. The ranking is used to determine which individuals should survive and propagate to the next generation and which individuals should be discarded or modified. Newer

individuals are produced by methods such as crossover and mutation operations to replace the bad solutions and create a new generation. Crossover operation combines the characteristics of two or more existing individuals to create a new individual. Mutation operation creates a new individual by modifying the genotype of an individual.

In a Genetic Algorithm, a random initial population is created and a metric is assigned to each individual of the population. The individual with the best metric is the best solution of the population of the current generation. Newer generations of populations are created from existing individuals using crossover and mutation operations and each individual is evaluated and assigned a metric. A best solution from the population of the latest generation is selected. The process of creating generations of population, evaluating them and selecting a best solution continues for a preset number of generations or until any other terminating condition is reached.

For the register allocation problem of the KITE architecture, a set of CSEs is treated as a phenotype. A vector that consists of single bits, with each bit corresponding to one CSE, is the genotype. The bits in the vector are randomly turned on or off (that is, set to 1 or 0 respectively). A CSE with its corresponding bit turned on in the vector will be stored in a temporary register so that it can be accessed until its reference count is zero. A CSE whose bit in the vector is turned off is recomputed every time it is referenced. The population of the genetic algorithm is made up of a number of CSE vectors with the CSE bits randomly turned on or off. A metric assigned to each individual is a combination of the maxlive and the instruction count required to evaluate the basic block. The metric is evaluated by doing register allocation of the sites and storing those CSEs whose bit in the vector is turned on. The individuals with lower maxlive and instruction count are assigned a lower metric. The individuals that fail the register allocation, because the set of CSEs turned on in the vector resulted in a higher

maxlive than the available number of registers, get the highest metric. For register allocation in the KITE architecture, the smaller the metric, the better.

3.3 Sethi-Ullman Numbering Extension

The nanocontrollers defined by the KITE architecture have N ≥ 1 general purpose registers, which are required by SUN algorithm. However, the nanocontrollers :

1. Do not support register-memory instructions, because there is no external memory. There are only 64 single-bit registers.
2. Use single-bit operations to implement all functions, thus requiring logic optimization techniques which naturally yield Directed Acyclic Graphs (DAGs) rather than trees.
3. Implement ternary, not binary operations.
4. Have issues that require considering the evaluation of multiple expressions as a single, integrated problem; even if the DAGs were trees, the ordering of the trees must be considered because the results from earlier computations are stored in registers.

The lack of register-memory instructions requires only a minor adjustment to the node labeling of SUN algorithm. However, the three other issues that are specified above are more difficult to resolve. There have been many attempts to extend SUN to handle optimal register allocation and instruction scheduling for DAGS [ApS86]. The fact that DAGs for nanocontroller programs are exceptionally large and complex makes the algorithm's execution time significant and yields a very small fraction of the DAGs for which special-case extensions of SUN can be applied. This research first tackles the problem of DAGs trees by converting the DAGs to trees. This is done by recomputing (or replicating) every CSE at every node where the CSE is referenced. Coincidentally, this process also reduces maxlive because it is not required to store any CSEs. This solution may seem extreme, but DAGs generally have an inherently higher maxlive than a tree. Given the extreme register pressure in the KITE architecture, it became necessary to focus first on reducing maxlive and only then to attempt to use

some of the benefits of CSE elimination. After tree generation, the next step is to find the optimum number of registers required to evaluate a node. This is done by using Sethi-Ullman Numbering and extending it to the ternary trees of nanocontroller blocks.

3.3.1 Tree Generation

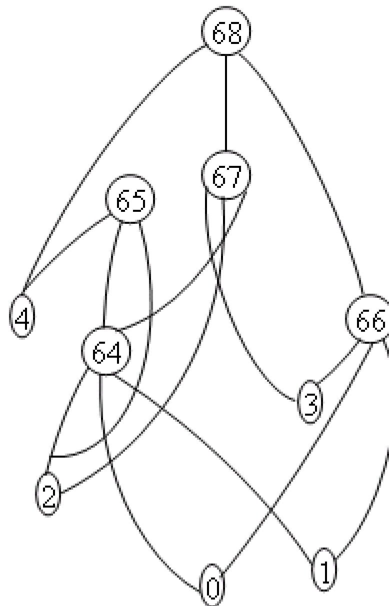
BitC compiler for KITE Nanocontroller architecture generates basic blocks that contain large and complex DAGs with many CSEs. The BitC compiler-generated DAGs contain nodes which represent *ite*s. To extend the SUN algorithm for nanocontroller generated basic blocks it is necessary to convert the *ite*-DAGs to a tree representation. All the *ite*s are first converted to *site*s by combining the *ite* operation with a store into a register operation. Each *site* corresponds to a node in the *ite*-DAG. The root node of every DAG corresponds to a *site* that represents the final store into a variable. All the interior nodes correspond to the temporary *site*s. The KITE hardware was originally designed to have 64 single bit registers. These 64 registers (numbered 0 – 63) are used for *ite*s 0 and 1 (ITE0 and ITE1) and the user variables. Hence the temporary *site*s are numbered starting from 64. After the *ite* to *site* conversion, a *site*-tree is generated by re-computing the CSE nodes. The leaf nodes of the site-tree correspond to either the *ite*s 0 and 1 or the initially defined user-variables.

Figure 3.2 shows sample code for the *ite*s generated by the BitC compiler and their corresponding *site* representation. Figure 3.3 shows the DAG representation of the *site*s of the sample code in Figure 3.2.

Figure 3.2 *ite* to *site* conversion

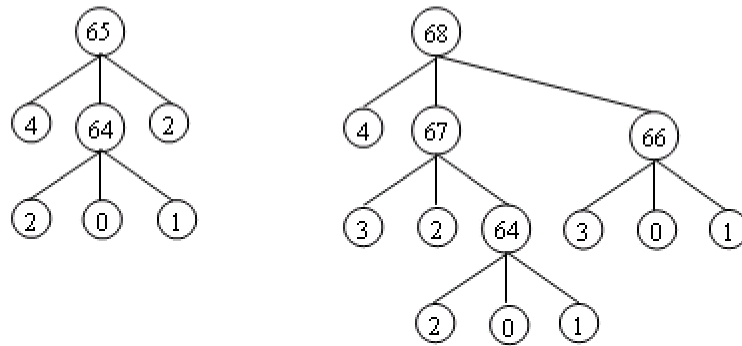
ite structure	→	*site* transformation	
ite 2 ? 0 : 1		*site 64:*	2 ? 0 : 1
ite 4 ? 64 : 2		*site 65:*	4 ? 64 : 2
ite 3 ? 0 : 1		*site 66:*	3 ? 0 : 1
ite 3 ? 2 : 64		*site 67:*	3 ? 2 : 64
ite 4 ? 67 : 66		*site 68:*	4 ? 67 : 66

Figure 3.3 DAG Representation of sites



A *site*-tree is generated from a DAG by replicating each common *site* at every node that references the common *site*. The DAG tree shown in Figure 3.3 consists of a node 64 that is referenced by node 65 and node 67. To convert the DAG to a tree representation, node 64 is replicated at each reference. Figure 3.4 shows the tree generated from the DAG in Figure 3.3.

Figure 3.4 Tree representation of sites



3.3.2 Extended Sethi-Ullman Numbering

The first step in tackling the register allocation problem is to reduce the high maxlive caused by the large and complex basic blocks which are in the form of DAGs. This is done by converting DAGs to trees by eliminating the common subexpressions of the DAG. The next step is to compute the optimum number of registers required for the trees. The three operand *ite* operation which is the main feature of KITE hardware and software results in ternary trees and hence the SUN algorithm, which focuses on the binary trees, has to be extended to ternary trees. The presence of three child nodes for all non-leaf nodes requires a more complex set of labeling rules than in SUN. Table 3.2 shows the labeling rules developed for BitC compiler generated basic blocks. Constant values 0 and 1, user-defined variables and any other input/output values are assumed to be already stored in registers and hence do not require any register allocation. The nodes corresponding to such pre-allocated values are represented as leaves of the tree. Hence, rule 1 of the extended SUN shown in Table 3.2 implies that the label (or the number of registers required during register allocation) for leaf nodes is always zero. Rule 2 of the extended SUN defines node labeling for intermediate nodes.

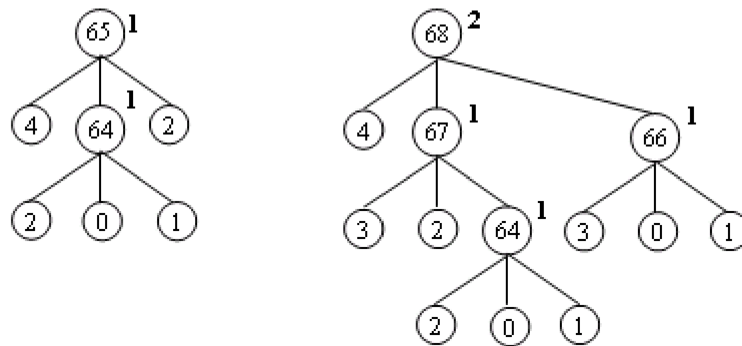
Table 3.2 Extended SUN Labeling Rules

Node Condition	Rule
1. η is a leaf	$L(\eta) = 0$;
2. η is a not a leaf	If η had descendants with labels l_1, l_2 and l_3 such that , $l_1 \geq l_2 \geq l_3$ a. $l_1 > l_2 > l_3$, $L(\eta) = l_1$; b. $l_1 > l_2 = l_3 = 0$, $L(\eta) = l_1$; c. $l_1 > l_2 = l_3 \neq 0$, $l_1 - l_2 = 1$, $L(\eta) = l_1 + 1$; d. $l_1 > l_2 = l_3 \neq 0$, $l_1 - l_2 > 1$, $L(\eta) = l_1$; e. $l_1 = l_2 > l_3$, $L(\eta) = l_1 + 1$; f. $l_1 = l_2 = l_3 \neq 0$, $L(\eta) = l_1 + 2$; g. $l_1 = l_2 = l_3 = 0$, $L(\eta) = 1$;

The two rules in Table 3.2 are defined for ternary trees and not DAGs that are produced by the BitC compiler. Although the rules for labeling the ternary trees of the KITE architecture are based on the labeling of binary trees of SUN, the rules of the extended SUN require more checks of the child node labels than in SUN. A generalization of extended SUN to label an n-ary tree (a tree with *n* child nodes) is discussed later in the chapter.

Figure 3.5 shows the node labeling of the ternary trees using the labeling rules of extended SUN that are defined Table 3.2. All the leaf nodes, indicated by node values less than 64 (nodes 0, 1, 2, 3 and 4) require no extra registers. Therefore, the figure shows no labels for the leaf nodes. The labels for all the intermediate nodes are calculated using the labeling rules.

Figure 3.5 Tree labeling using extended SUN



3.3.3 Register Allocation using SUN

Algorithm 1 discussed in section 3.3.3.1 evaluates the number of registers and instructions required for each store into a variable in a basic block. The root node of every tree represents a final store into a user-defined variable. Therefore the root node (or the store node) of every tree has a node number less than 64. Starting from node 64 are all the intermediate *site*s that actually require register allocation. Tree evaluation proceeds from the top, starting from the root node, to the leaf nodes. Each node's descendants are evaluated recursively. Label for each node, which represents the number of registers required at the time of evaluation of that node, is generated as the tree is traversed from top to bottom.

3.3.3.1 Algorithm 1

For every node,

1. If the node is not a store, skip to next node.
2. If the node is a store, examine its sub-tree as follows:
   a. For each child node of the current node:
      i. Evaluate the number of registers required for the node by applying the labeling rules defined in extended SUN to the node's sub-tree.
      ii. Track the node's reference count and increment it each time the node is referenced.
      iii. Evaluate the number of instructions required (called *depth*) for the node. The depth of any node is the sum of the depths of the descendants.

b. Calculate the number of registers and depth for the store node, again using the labeling rules of extended SUN.

Algorithm 1 evaluates a single basic block. Registers for each intermediate node are allocated during the evaluation of the node. Instruction count is also evaluated by adding all the nodes evaluated during the tree traversal. A reference count for each node is maintained to keep track of the number of times each node is referenced by any other node. Any node that has a reference count greater than 1 is a *common subexpression*. As mentioned in section 3.3.1, a tree is generated by replicating the common subexpressions. Every node that is a common subexpression is expanded to form a complete sub-tree rather than storing the node's value. Algorithm 1 merely updates the reference count of each node of the basic block. Common subexpressions are not stored temporarily to be readily accessed when referenced again. Instead, each common subexpression node is entirely re-evaluated every time it is referenced. The advantage with this approach is that once a node has been evaluated, all the temporary registers used to evaluate it, that is the registers for the node's sub-tree, are freed and the risk of running out of registers is extremely reduced. The disadvantage is that evaluation of the nodes, including the re-evaluation of the common subexpressions each time they are referenced, results in instruction count increasing exponentially. A simple 8-bit square (a^2) operation typically required less than 20 registers whereas the instruction count is in the order of thousands.

Algorithm 1 results in the register usage being much lower than the total number of registers available, whereas the instruction count is exceptionally high. The high instruction count could be potentially reduced by using the unused registers to store the common subexpressions. All the common subexpressions of a basic block cannot be stored as that would increase the maxlive and defeat the purpose of this research. However, selectively storing only a few common subexpressions such that maxlive does not exceed the register limit would

certainly reduce the instruction count. The problem of selecting a few common subexpressions to store is not trivial. Exhaustive search of all the combinations may be a viable option for smaller sets, but the search space grows exponentially as the number of common subexpressions grows. For example, an 8-bit, two-operand multiplication generated 939 common subexpressions; this yields 2^{939} different combinations to search. An alternative approach that uses a genetic algorithm to tackle this problem is described in the next section.

3.4 Extended Sethi - Ullman Numbering with Genetic Algorithm

Although Algorithm 1 solves the problem of large maxlive of KITE architecture's nanocontroller code by eliminating temporary storage of common subexpressions, the number of instructions executed per block increases significantly because of the conversion of DAGs to trees. It is possible to reduce the total number of instructions executed per block by storing a subset of the total number of common subexpressions generated per block temporarily. This section describes methods that are used to select a set of common subexpressions that can be stored for multiple references while remaining within the register limit, thus reducing the instruction count.

3.4.1 Extended SUN using Genetic Algorithm

Algorithm 1 helps identify the common subexpressions of a basic block by tracking the reference count of all the nodes. Common subexpressions are the nodes with a reference count greater than 1. To reduce the total instruction count per block, it is necessary to selectively store a subset of common subexpressions during register allocation, or possibly all of them, and stay with in the register limit. With very large basic blocks generated by the KITE architecture's BitC compiler, the probability of large sets of common subexpressions is very high. Algorithm 2 extends the maxlive reduction developed in Algorithm 1 to generate a set of common subexpressions that can be stored by using a genetic algorithm. The genetic algorithm may not produce the most optimal result for reducing maxlive and instruction count. However, any good result that is produced by a

genetic algorithm results in an instruction count that is smaller than that of Algorithm 1.

3.4.1.1 Algorithm 2

1. Apply rules 1 and 2 of Algorithm 1 to calculate the reference counts of all the nodes and determine the set of common subexpressions.
2. Create a vector of single bits, each bit corresponding to a common subexpression. The vector size is equal to the total number of common subexpressions. The vector represents the genotype of the population of the genetic algorithm.
3. Create an initial population of common subexpression vectors. This is done by randomly turning the bits in the genotype vector on or off for each member of the population.
4. If the node is a store, for each child node:
   a. Compute the registers required:
      i. Node is a CSE with its vector bit turned and is referenced for the very first time - compute the number of registers required for the node by applying the labeling rules defined in extended SUN and allocate a register for the node.
      ii. Node is a CSE with its vector bit turned one and the current reference is greater than 1 – this indicates that this node has already been evaluated and the result stored in a register. Therefore, no extra evaluation is required and the number of registers required is zero.
      iii. Node is a CSE with its vector bit turned off - the node is not stored in a register and has to be re-evaluated. The total number of registers required will be computed using extended SUN.
   b. If number of registers required for the node is greater than the register limit of the system go to step 5.a.
   c. Evaluate the number of instructions required (or the *depth*) for the node.
      i. Node is a CSE with its vector bit turned off and the current reference to the node is 1 – this indicates that this node has not been previously

evaluated. Compute the depth of the node, which is the sum of the depths of the descendants.

ii. Node is a CSE with its vector bit turned on and the current reference count is greater than 1 – this indicates that the node has been evaluated and the result is stored in a register. Therefore, the depth for the node for all references greater than one is 1.

iii. Node is a CSE with its vector bit turned off - this node has to be re-valuated. There for the total number of instructions required to evaluate the node is the sum of the all the descendants of the node.

d. Go to step 5.b.

5. Evaluate the metric of each individual of the population.
   a. Assign a large metric to the current population such that it is proportional to the number by which the register limit is exceeded.
   b. Calculate the metric such that it is a function of the number of register and number of instructions required to evaluate each basic block.
6. Repeat steps (4) – (5) for the entire population.
7. Sort the population based on the metric of each individual and select the best result. In this case, it is the one with the lowest value of the metric.
8. Create newer population members by applying crossover and mutation operations.
9. Repeat steps (4) – (8) for a predetermined set of generations. The final solution is the one with a lower metric among the best results of all generations.

Algorithm 2 is a two pass scan of a basic block. Step 1 is the first pass in which the number of registers and instructions required for each store in a basic block are calculated conservatively. The first pass, which is Algorithm 1, is done to compute the reference counts of all the nodes and identify the common subexpressions (that is, nodes with reference count greater than 1). The rest of Algorithm 2 constitutes the second pass in which register allocation is done by using a genetic algorithm. A bit vector corresponds to a list of all the common

subexpression nodes in the basic block. If a bit is turned on, the corresponding common subexpression node is evaluated the first time and stored in a register. If a bit is turned off, the common subexpression node is evaluated every time it is referenced and is not stored in a register temporarily. An initial population of bit vectors is created by turning the bits on or off for every individual vector. Step 4 reflects a big change from Algorithm 1, where the common subexpressions are selectively stored after they are evaluated the first time, to be used for all subsequent references. Even though storing the common subexpressions increases maxlive, in some cases it may possible that the number of registers required decreases. The reduction can be attributed to the fact that in the case of a common sub-expression with a large tree, storing the CSE in a register may reduce the number of registers required to evaluate its large sub-tree, thus reducing the maxlive. The total number of instructions required for each node that accesses a stored common subexpression is also reduced. The fitness value or the metric of each individual of the population is a function of the number of registers and the number of instructions required for each basic block. Step 4b implies that if the number of registers required to evaluate a node exceeds the register limit, the current population member cannot be used and is regarded as a bad solution. Therefore, a large constant metric value, which is proportional to the number of registers by which it exceeds the register limit, is assigned to a bad solution.

After the metric is assigned to all individuals, the population is approximately sorted. A subset of the population that will survive and propagate to the next generation is determined by the sorted order. Therefore, the sort is deliberately made approximate and stochastic to ensure that the population maintains an acceptable level of genetic variety. It is necessary to have a population with genetic variety so that the results are not skewed towards one direction because of similar population members. The members of the population that are not selected to survive are replaced by newer members created using crossover and mutation operations. For the register allocation of nanocontrollers,

two-parent crossover is implemented, in which a new common subexpression bit vector is created by combining the bit vectors of two existing individuals. Mutation is implemented by randomly changing some bit values of an existing individual.

3.4.2 Extended SUN: Tree Re-Ordering

The large basic blocks with large maxlive that are generated by the BitC compiler made it essential to find a register allocation solution by reducing maxlive. Algorithm 1 converted DAGs generated by the BitC compiler to trees in order to minimize maxlive. Algorithm 1 also resulted in many unused registered and increased instruction count because of re-evaluation of all common subexpression at each reference. Algorithm 2 selectively stored common subexpressions to reduce the instruction count. Algorithms 1 and 2 perform a straight-line tree evaluation, which means each tree is evaluated in the order of its appearance in the basic block. The next step is to find a tree evaluation order that may result in a solution that is better than the one generated by Algorithm 2. The process of finding a tree evaluation order is not trivial because of the large code blocks generated by BitC compiler. Therefore a genetic algorithm is used for finding a tree-evaluation order as in Algorithm 2.

The basic rules for register allocation and instruction count are the same as in Algorithm 2. Algorithm 3 extends Algorithm 2 to select a tree evaluation order using a genetic algorithm. Two Genetic Algorithms must be applied simultaneously to two different structures namely – (1) the set of common subexpressions and (2) the set of trees. The root node of each tree is a final store into a register. The final stores are indicated by node numbers that are less than 64.

3.4.2.1 Algorithm 3

1. Apply rules 1 and 2 of Algorithm 1 to calculate the reference counts of all the nodes and determine the set of common subexpressions.
2. Create an initial population for the final stores. This is done by randomly assigning priority to each store and sorting the stores in decreasing order of the priority, which means that each tree, now called a priority tree, is evaluated in the order of its priority.
3. Create an initial population for the common subexpressions. This is done by randomly turning the bits of the genotype on or off for each member of the population.
4. For every child node of every store node in the priority tree
   a. Compute the registers required:
      i. Node is a CSE with its vector bit turned and is referenced for the very first time - compute the number of registers required for the node by applying the labeling rules defined in extended SUN and allocate a register for the node.
      ii. Node is a CSE with its vector bit turned one and the current reference is greater than 1 – this indicates that this node has already been evaluated and the result stored in a register. Therefore, no extra evaluation is required and the number of registers required is zero.
      iii. Node is a CSE with its vector bit turned off - the node is not stored in a register and has to be re-evaluated. Therefore, the total number of registers required will be computed using extended SUN.
   b. If number of registers required for the node is greater than the register limit of the system go to step 5.a.
   c. Evaluate the number of instructions required (or the *depth*) for the node.
      i. Node is a CSE with its vector bit turned off and the current reference to the node is 1 – this indicates that this node has not been previously evaluated. Compute the depth of the node, which is the sum of the depths of the descendants.

ii. Node is a CSE with its vector bit turned on and the current reference count is greater than 1 – this indicates that the node has been evaluated and the result is stored in a register. Therefore, the depth for the node for all references greater than one is 1.

iii. Node is a CSE with its vector bit turned off - this node has to be re-valuated. There for the total number of instructions required to evaluate the node is the sum of the all the descendants of the node.

d. Go to step 5.b.

5. Evaluate the metric of each individual of the population.
   a. Assign a large metric that is proportional to the number of registers exceeded to this member.
   b. The metric is a function of the number of register and instructions required to evaluate each basic block.
6. Repeat steps (4) – (7) for the entire population.
7. Sort the population based on the metric value of each individual.
8. Apply crossover and mutation operations to create new individuals in the populations of trees and the common subexpression bit vectors.
9. Repeat steps (4) – (10) for a preset number of generations.

3.5 Generalization of Extended SUN for of n-tuples

The Sethi-Ullman Numbering method described in section 3.1 specifies labeling rules for arithmetic expressions represented by binary trees. An extended set of labeling rules is described in section 3.3.2 for ternary trees generated by the BitC compiler of KITE Nanocontroller architecture. A comparison of the labeling rules of SUN and extended SUN shows an increase in the number of rules as the order of the trees increases from binary to ternary. As the order of the trees increases further, it becomes difficult to develop a set of labeling rules that is comprehensive. Section 3.5.1 describes a generic node labeling method for an *n-ary* tree (a tree with *n* child nodes).

3.5.1 Labeling Rules for an n-ary tree

Table 3.3 contains the labeling rules an n-ary tree. Rule 1 of Table 3.3 is the same as that of SUN in Table 3.1 and of extended SUN in Table 3.2, that is, the leaf nodes do not require any extra registers because they are pre-stored in registers as user defined variables or constants. Therefore, the label of leaf nodes is zero. Rule 2.a implies that the label of a node is 1 if all the children of the node have a label of 0. Rule 2.b implies that the label of a node whose descendants have unique labels is equal to the largest label of all its descendants. Rules 2.c and 2.d do not assign a label to any node and are used to determine a label that is used as an intermediate value. When any 2 descendants are compared, if the two nodes have the same label, the number of registers required for the evaluation of the two nodes is one more than either label. If the two nodes have different labels, the number of registers required for the evaluation of the two nodes is the larger of the two labels. Intermediate labels are generated by comparing iteratively all the child nodes of the current node. The iterative comparison generates a set of unique labels that is sorted. Rule 2.b is applied to the sorted set of descendant labels to assign a label to the parent node.

Table 3.3 Labeling rules for a generic tree

Node Condition	Rule
1. η is a leaf	$L(\eta) = 0$;
2. η is a not a leaf	If η had n descendants with labels l_1, l_2, l_3 … l_n a. $l_1 = l_2 = l_3 \ldots = l_n = 0$, $L(\eta) = 1$ b. $l_1 > l_2 > l_3 \ldots > l_n$, $L(\eta) = l_1$ For any 2 descendants with labels l_r and l_{r+1} c. $l_r = l_{r+1}$, $L(l) = l_r + 1$ d. $l_r > l_{r+1}$, $L(l) = l_r$, where $L(l)$ is the number of registers required for the 2 descendants only

3.5.2 Register Allocation Algorithm for an n-ary tree

The register allocation for an n-ary tree uses the labeling rules described in Table 3.3. The register allocation algorithm is applicable to trees. Therefore, all the DAGs must be converted to trees by replicating all the common subexpression nodes at every point of reference.

3.5.2.1 Algorithm 4

For every node

1. If the node is a final store go to step 2, else skip to the next node.
2. Evaluate the number of registers required for the node by applying the labeling rules:
   a. If the node is a leaf return zero.
   b. If the node has descendants, for each child node, go to step 2.
   c. If all the descendants are evaluated, generate a sorted set of descendant labels. Temporary values are generated by comparing every descendant label to its adjacent label recursively by applying rules 2.c and 2.d.
   d. The number of registers required for the node is the largest of the sorted descendant labels.

As in Algorithm 1, common subexpressions are not stored temporarily using Algorithm 4. All the common subexpression nodes are re-evaluated at every reference. Algorithm 4 reduces maxlive and results in fewer registers being used for register allocation but increases the number of instructions. The number of instructions can be reduced by selectively storing the common nodes temporarily while staying within the register limit. This can be done by using a genetic algorithm similar to that used in Algorithm 2.

Although the labeling rules for a generic n-ary tree defined in Table 3.3 have been developed, the register allocation algorithm for an n-ary tree has not been implemented because the main goal of this research was register allocation

for the ternary trees generated by the BitC compiler of the KITE architecture as described in Algorithms 1, 2 and 3.

4 Results

Algorithms 1, 2 and 3 were analyzed by executing three trials. Each trial consisted of executing a BitC program three times using 2-bit, 4-bit and 6-bit operands (thus, a total of 9 different tests). Each program was a mix of simple addition, subtraction, multiplication and division operations. For algorithms 2 and 3 that used genetic algorithms, the total number of generations in each was 10 and the maximum population count was 1000. Each genetic algorithm performed crossover operation on 300 individuals and mutation operation on 200 individuals.

Algorithm 1, which converts DAGs to trees, was used to calculate the total number of CSEs in every trial. The root node of each tree represents a final store of the result of an arithmetic operation in every trial. The CSEs identified in Algorithm 1 are used in Algorithm2 to generate a genome represented by a bit-vector. Each bit in the vector corresponds to a CSE. Therefore the size of the bit vector is equal to the number of CSEs in each trial. In Algorithm 3, the genome consists of an array of trees. Each element of the array corresponds to a tree. A random priority assigned to each tree element in the array determines the order of tree evaluation. In each test run, maxlive was computed by performing actual register allocation, in addition to determining the number of registers required for tree evaluation by node labeling using extended SUN. Additionally, *site*s (or the instructions executed) for each tree are also calculated.

4.1 Results

The following sections plot bar graphs for the 27 test runs (that is, nine BitC programs executed three times each using extended SUN, extended SUN using Genetic Algorithm and extended SUN using Tree Re-Ordering algorithms). Two sets of graphs are plotted per sample program – one for maxlive and the other for the *site*s per block. Each graph shows a comparison of the three algorithms. In each graph, the Y-Axis consists of 3 groups of bars. Each group represents the execution of the three algorithms for one set of operand sizes.

Group 1 corresponds to a BitC program with 2-bit operands. Group 2 corresponds to the same BitC program with 4-bit operand and Group 3 corresponds to the same BitC program with 6-bit operands. The X-axis represents maxlive or *site*s. Algorithm 1 is represented as 'SUN' series, Algorithm 2 is represented as 'SUN-GA' series and Algorithm 3 is represented as 'SUN-GA-TREE-REORDER' series.

4.1.1 Trial 1

Trial 1 consists of the following BitC program:

```
int: x a, b, c;
c = a * c;
a = a + c;
```

In the program, *x* may be for 2, 4 or 6 indicating 2-bit, 4-bit or 6-bit operands respectively for a, b and c. In the 2-bit run, the total number of CSEs generated was 2. In the 4-bit run, the total number of CSEs generated was 34. In the 6-bit run, the total number of CSEs was 324.

Figure 4.1 and Figure 4.2 show the bar graphs for the maxlive and *site*s respectively generated for the BitC program of trial 1. Group 1 shows the results of the three algorithms, Algorithm 1, Algorithm 2 and Algorithm3, for 2-bit operands. Group 2 shows the results of the three algorithms for 4-bit operands. Group 3 shows the results of the three algorithms for 6-bit operands. For Trial 1, as the CSEs were turned on selectively in the SUN-GA and SUN-GA-TREE-REORDER series, maxlive increased whereas *site*s decreased when compared to the SUN series where no CSEs were stored.

In Group 1, the maxlive for SUN is 33.3% less than that of SUN-GA and SUN-GA-TREE-REORDER series whereas the *site*s for SUN-GA and SUN-GA-TREE-REORDER were 25% less than in the SUN series. The difference in the results is not as pronounced because the number of CSEs for Group 1 is only 2. In Group 2, the maxlive for SUN is 85% less than in SUN-GA and 79% less than

in SUN-GA-TREE-REORDER series. The *site*s in SUN-GA and SUN-GA-TREE-REORDER series are 71% and 69% less respectively than in the SUN series. In Group 3, the maxlive in SUN series is 81% less than in SUN-GA and SUN-GA-TREE-REORDER series whereas *site*s in SUN-GA and SUN-GA-TREE-REORDER series are only 49% less than the SUN case.

Figure 4.1 Trial 1 - maxlive

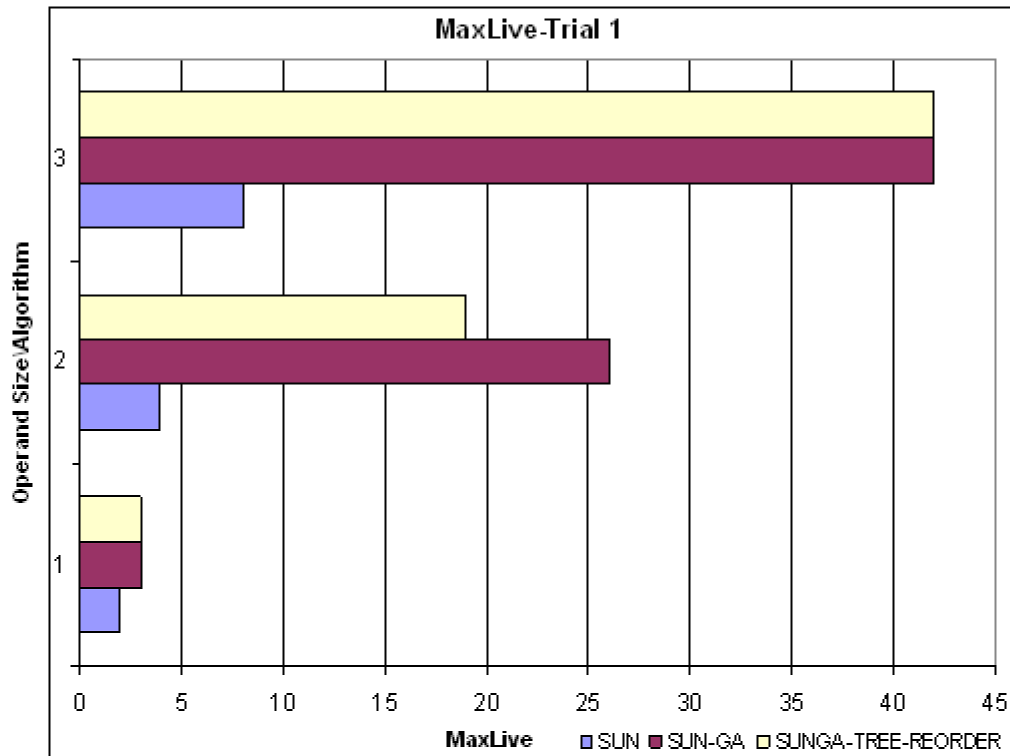
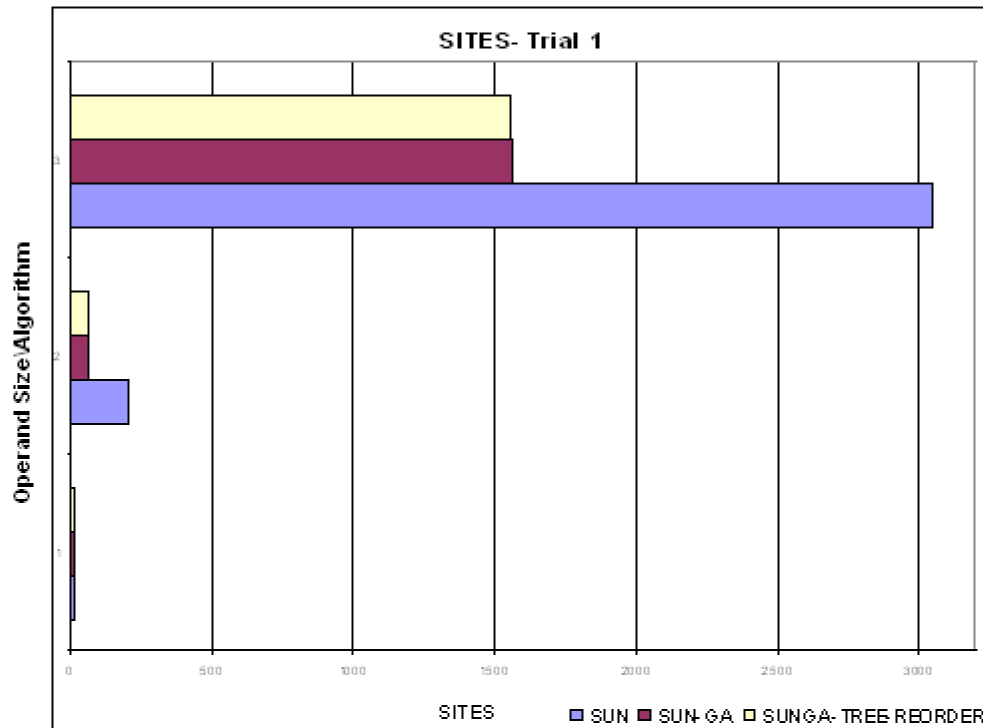


Figure 4.2 Trial 1 - *site*s



4.1.2 Trial 2

Trial 1 consists of the following BitC program:

int: *x* a, b, c;

a = a * c;

b = b * c;

In the program, *x* may be for 2, 4 or 6 indicating 2-bit, 4-bit or 6-bit operands respectively for a, b and c. In the 2-bit run, the total number of CSEs generated was 0. In the 4-bit run, the total number of CSEs generated was 22. In the 6-bit run, the total number of CSEs generated increased significantly to 322.

Figure 4.3 and Figure 4.4 show the bar graphs for the maxlive and *site*s generated. Group 1 shows the results of the three algorithms for 2-bit operands. Group 2 shows the results of the three algorithms for 4-bit operands. Group 3 shows the results of the three algorithms for 6-bit operands. As in Trial 1, when

the CSEs were turned on selectively in the SUN-GA and SUN-GA-TREE-REORDER series, maxlive increased whereas *site*s decreased again when compared to the SUN series where no CSEs were stored.

Figure 4.3 Trial 2 - maxlive

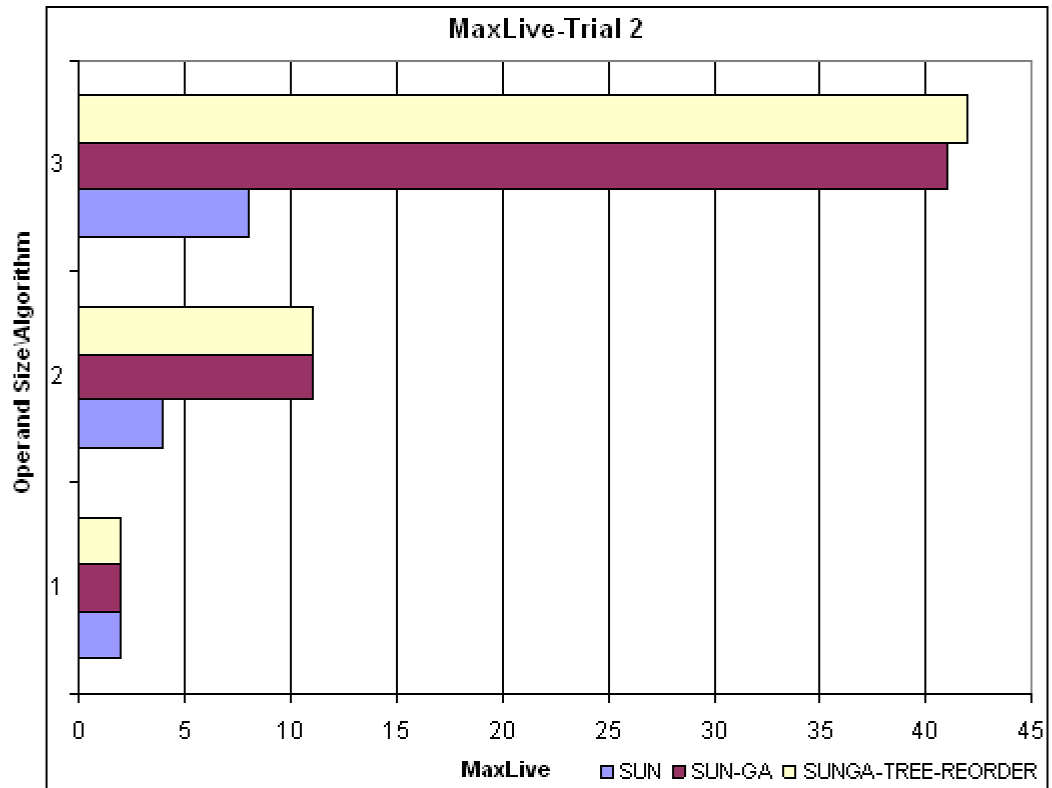
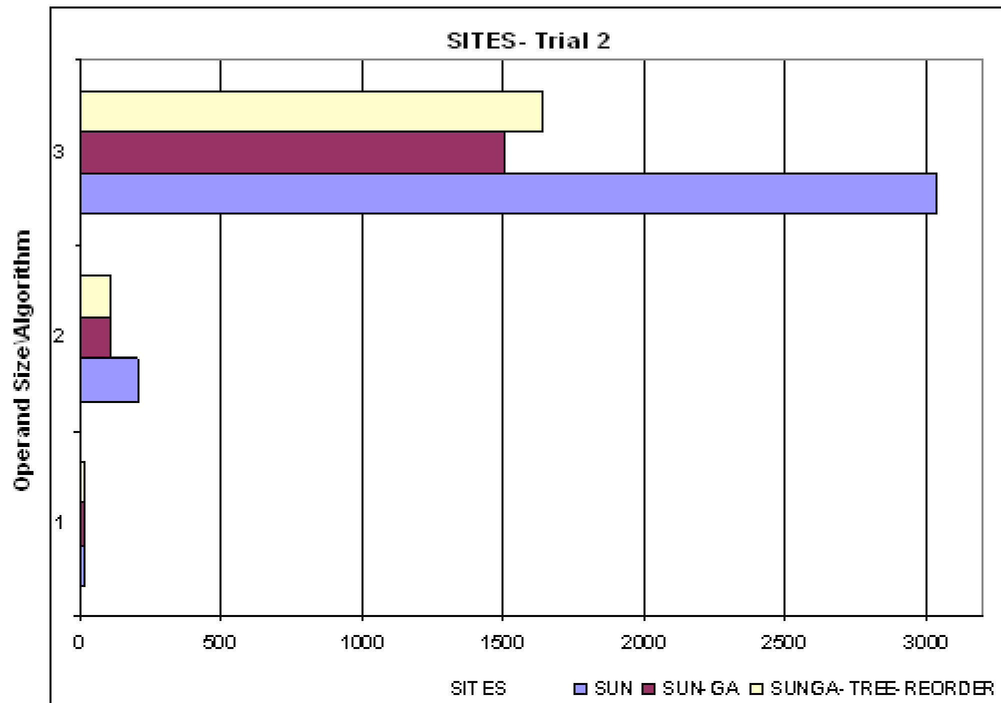


Figure 4.4 Trial 2 -*site*s



In Group 1 with zero CSEs, the results for maxlive and *site*s results are identical in the SUN, SUN-GA and the SUN-GA-TREE-REORDER series. In Group 2, the maxlive for SUN is 64% less than the SUN-GA and SUN-GA-TREE-REORDER series whereas the *site*s for SUN-GA and SUN-GA-TREE-REORDER are approximately 45% less than in the SUN series. Group 2 shows almost identical results in the SUN-GA and the SUN-GA-TREE-REORDER cases for maxlive and the *site*s. This is not unexpected because re-ordering of the tree evaluation does not always yield a better result as the results depend on the quality of the genetic algorithms. In Group 3, the maxlive for the SUN series is 80% less than in SUN-GA series and 81% less than in SUN-GA-TREE-REORDER series. The *site*s for SUN-GA and SUN-GA-TREE-REORDER are only 50% and 46% less than the SUN series. It is interesting to note that in Group 3 both maxlive and *site*s are higher in the SUN-GA-TREE-REORDER series when compared to the SUN-GA case which is not surprising. The tree reordering increases the search space significantly and makes the genetic algorithm converge slowly, so a superior result may not be produced in the time

allotted. It should also be noted that when the CSEs increase greatly, a larger percentage increase in maxlive does not necessarily decrease the *site*s by a similar margin.

4.1.3 Trial 3

Trial 1 consists of the following BitC program:

int : *x* a, b, c, d;
c = a / b + d;

In the program, *x* indicates the bit size of the operands *a*, *b*, *c* and *d*. The value of x was chosen to be 2, 4 and 6 in three different runs of the program, each run executing modified SUN, modified SUN with GA and modified SUN with tree reorder. Figure 4.5 and Figure 4.6 show the bar graphs for the maxlive and *site*s generated. Each bar represents the program execution for a specific operand size and a specific algorithm.

In the 2-bit run, the total number of CSEs generated was 8. In the 4-bit run, the total number of CSEs was 132. In the 6-bit run, the total number of CSEs generated was 1158. The CSEs generated in each case are greater in number when compared with the corresponding runs in Trial 1 and Trial 2 because a division operation, like that in Trial 3, increases the total number of *ite*s, *site*s and hence the CSEs. Figure 4.5 and Figure 4.6 shows yet again that larger the number of CSEs, higher is the maxlive and *site*s. The plots show mixed results for the 3 groups. The total *site*s for the modified SUN case in Group 3, the 6-bit run, is nearly 20,000 whereas in the Group 1, the 2-bit run with modified SUN applied, the total *site*s is 40. This variation makes the plots in Group1 nearly invisible.

Figure 4.5 Trial 3 - maxlive

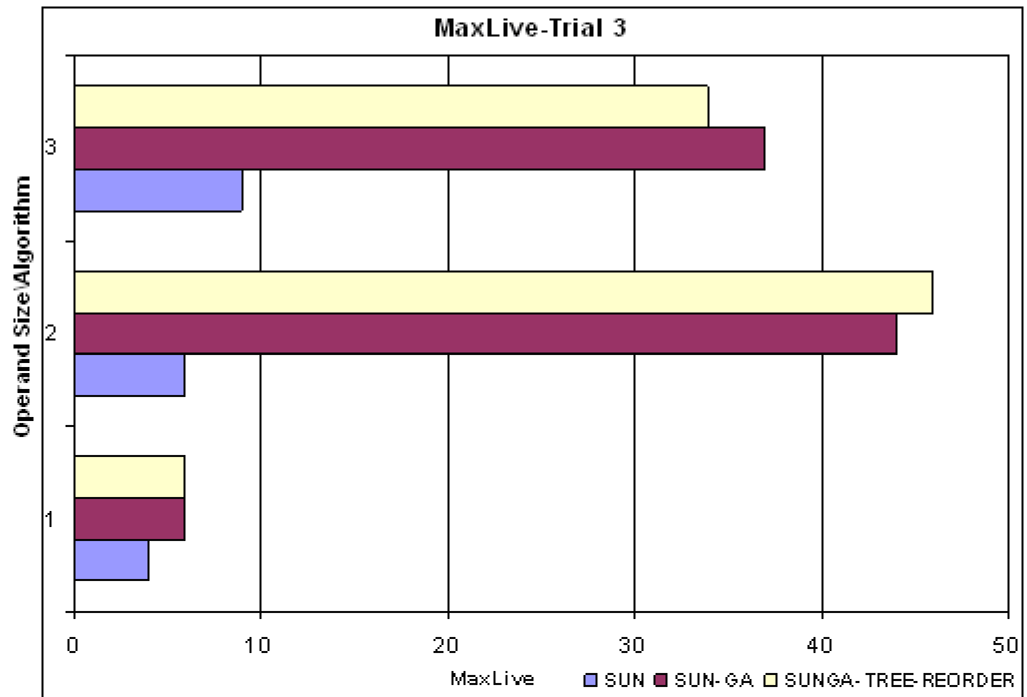
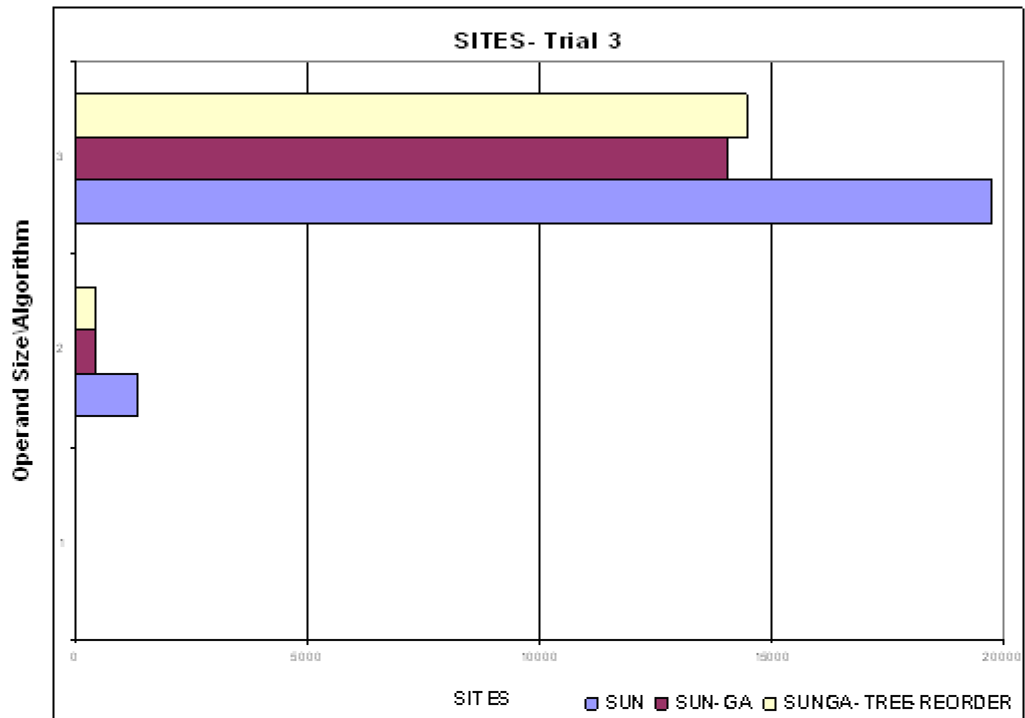


Figure 4.6 Trial 3 - *site*s



In Group 1 with 8 CSEs, the maxlive for the SUN case is 33.3% less than that of SUN-GA and SUN-GA-TREE-REORDER cases. The *site*s in the SUN-GA and SUN-GA-TREE-REORDER cases are 55% of the SUN case. This result for maxlive in Group 1 of Trial 2 is similar to the result for Group 1 in Trial 1. In Group 2, the maxlive in the SUN case is 86% less than the SUN-GA and SUN-GA-TREE-REORDER cases. The total *site*s in the SUN-GA and SUN-GA-TREE-REORDER are around 45% less than in the SUN case. The results for maxlive and *site*s in Group 2, with 4-bit operands, are almost identical in the SUN-GA and SUN-GA-TREE-REORDER cases. This is not unexpected because the result of tree re-ordering will not always yield a better result and depends greatly on the quality of the genetic algorithm. In Group 3, the maxlive in the SUN case is 80% less than in SUN-GA case and 81% less than in SUN-GA-TREE-REORDER case. The total *site*s in SUN-GA case and SUN-GA-TREE-REORDER case are only about 50% and 46% less than in the SUN case. It is interesting to note that in Group 3 both maxlive and the total *site*s are higher for the SUN-GA-TREE-REORDER case than the SUN-GA case. This is also not unexpected because the genetic algorithms used in the two algorithms will not always converge to the best possible solution. Also, as the CSEs increase, an increase in maxlive will not necessarily result in a large decrease of the total *site*s.

4.2 Effect of increasing available registers on CSEs and SITES

The number of registers available for KITE architecture was 64. By increasing the available registers it is possible to increase the number of CSEs turned on thereby decreasing the number of instructions executed per basic block. This section explores the effect of increasing the number of registers on the CSEs and total *site*s in the SUN-GA and SUN-GA-TREE-REORDER cases.

Figure 4.7 shows the graph for the number of CSEs that can be turned on as the register limit increases. After the initial rise, it is seen that the number of CSEs that can be turned on (or stored temporarily) remains constant. The results of Figure 4.7 are obtained for the modified SUN algorithm with a GA.

Figure 4.7 Effect on register limit on CSEs - SUN-GA

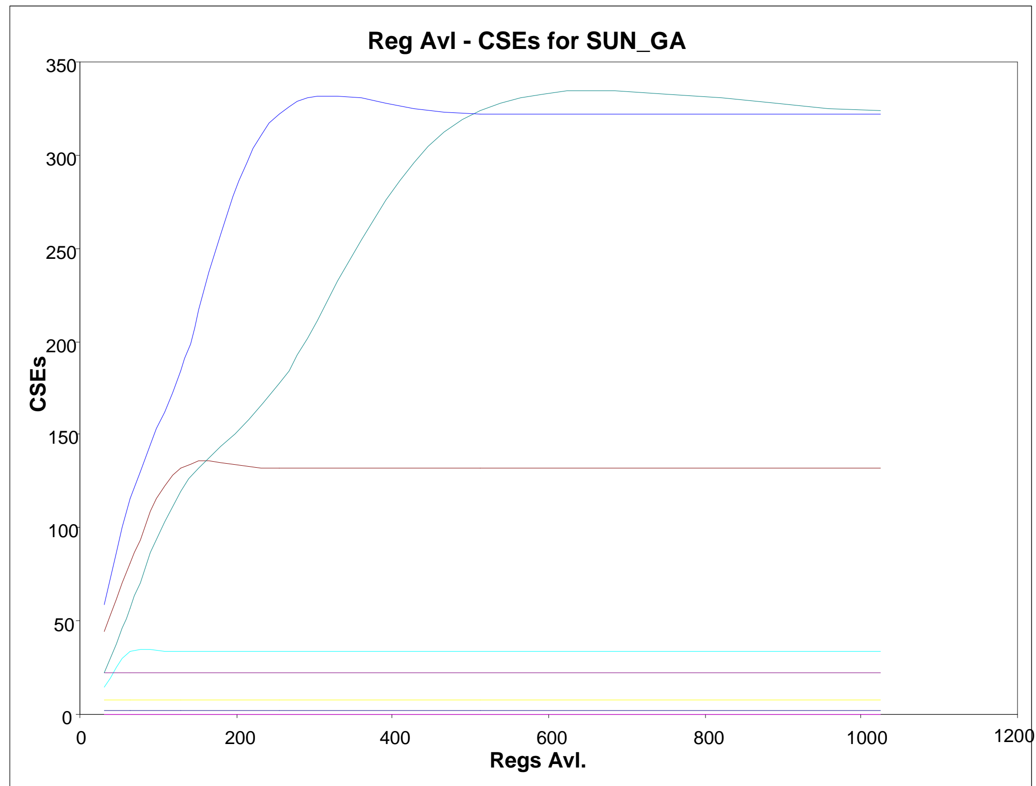


Figure 4.8 shows the graph for the number of CSEs that can be turned on as the available registers increase. As Figure 4.7, after the initial rise, the number of CSEs that can be turned on (or stored temporarily) remains constant. The results in Figure 4.8 are obtained by applying the for the modified SUN algorithm after applying the GA with reordered tree execution.

Figure 4.8 Effect of register limit on CSEs – SUN-GA-TREEREORDER

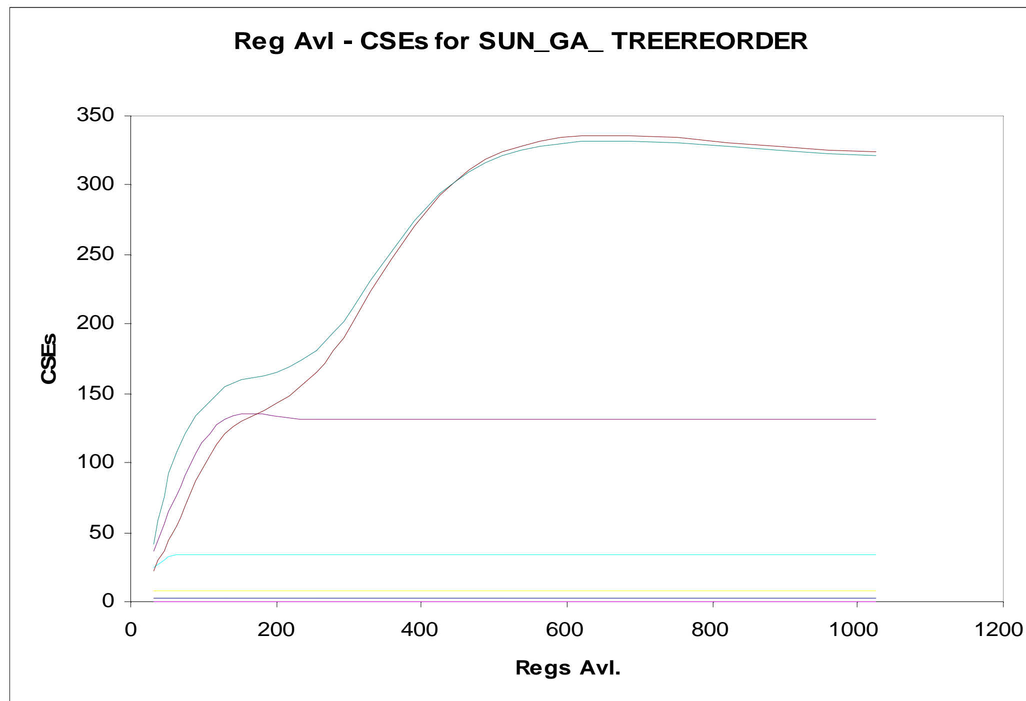


Figure 4.9 and Figure 4.10 show the graphs for the total *site*s executed as the register limit increases for the modified SUN algorithm after applying the GA and the reordered tree execution algorithms respectively. Both graphs show that as the number of registers available increases, the total number of *site*s that have to be executed per basic block decreases. The decrease in the total *site*s is a result of an increase in the number of registers available for storing CSEs temporarily. In the instances where the CSE count is low, as in the 2-bit runs of trials 1, 2 and 3, an increase in register limit has limited or almost no effect.

Figure 4.9 Effect of register limit on sites - SUN-GA

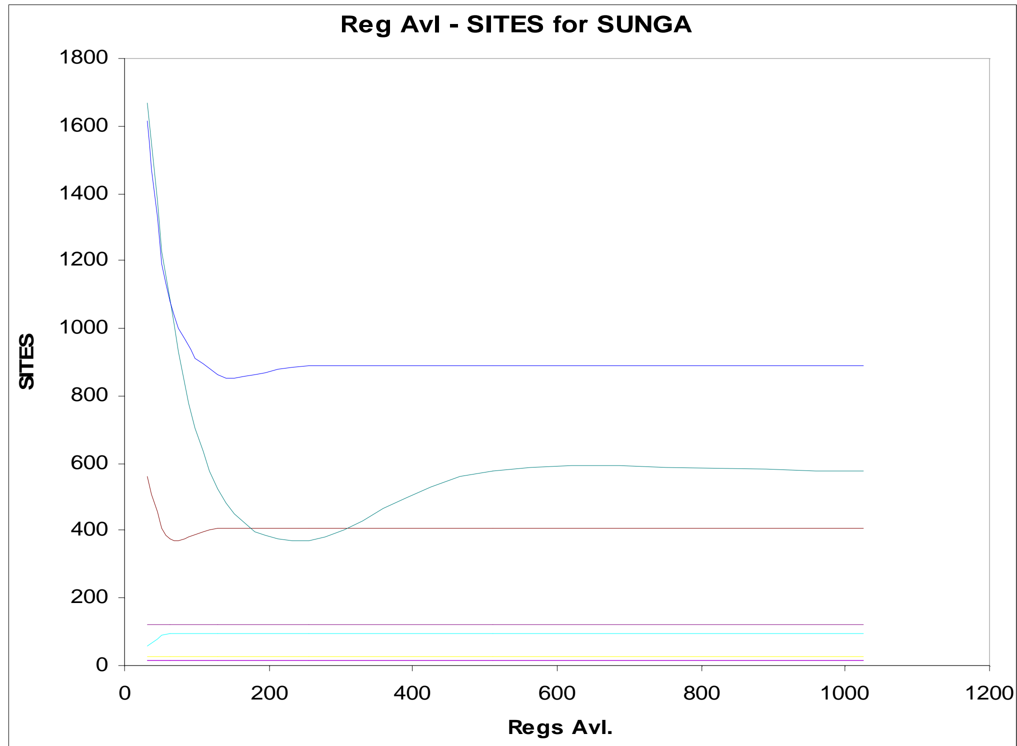
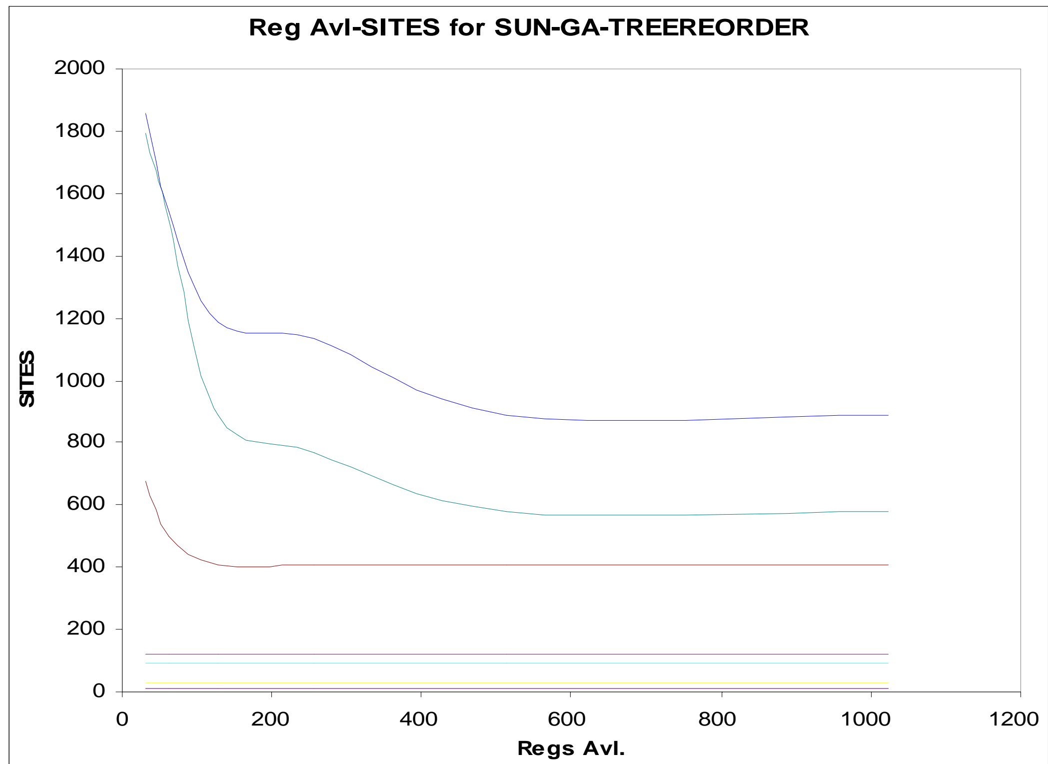


Figure 4.10 Effect of register limit on sites - SUN-GA-TREEREORDER



5 Conclusions

The KITE architecture achieved low controller hardware complexity by eliminating data memory and a severely limited register file. The bit level operations that have to be executed on the hardware required bit-level code generated by the BitC compiler. The code generation resulted in large and complex basic blocks. Existing methods for register allocation proved insufficient because of the severe hardware limitation and the large maxlive that resulted from the large basic blocks. This project realized the goal of finding a register allocation method for the KITE Nanocontroller architecture without which the KITE architecture's hardware minimization could not be achieved

Sethi-Ullman numbering, a popular register allocation method developed in 1970, generated optimal code for arithmetic expressions expressed in the form of binary trees. Sethi-Ullman numbering could not be used for DAGs that are primarily generated by many code generation methods. Many heuristics that were proposed to apply Sethi-Ullam numbering to DAGs concentrated on minimizing the register spill-cost. KITE architecture has no external data memory. Therefore, such heuristics could not be applied to the KITE architecture because of the lack of register-memory operations. Applying Sethi-Ullman numbering to DAGs achieved in this project by converting DAGs to trees. The conversion of DAGs to trees also reduced the maxlive of a basic block, thus reducing the probability of a register-spill. This project developed labeling rules for ternary trees generated by the BitC compiler by extending the labeling rules of Sethi-Ullman numbering for binary trees. Genetic algorithms were used to further optimize the results of register allocation using the extended Sethi-Ullman numbering. The register allocation algorithms assume no spill and do not require any external data memory. The solutions that require a register spill are discarded. Node labeling rules were also developed for a generic n-ary tree.

5.1 Application to other architectures

This project concentrated on maxlive reduction for basic blocks consisting of ternary operations generated by the BitC compiler of the KITE architecture. Conventional architectures do not execute instructions at single bit-levels and also deal with binary operations. However, the algorithms developed in this research can be used by existing architectures. The algorithms developed in this research were also generalized to be applied to n-ary trees; therefore, any special applications that implement a non-binary approach may use the node labeling rules for n-ary trees.

5.2 Future Work

The algorithms developed in this project are applicable for a single basic block. Future work may extend register allocation method across multiple blocks. The register allocation method may be applied to the KITE architecture hardware which was not yet developed during the compiler development. It should also be noted that the genetic algorithms used in this project for selecting the CSEs to be enabled and for selecting a tree evaluation order were simplistic because of time considerations. Therefore, the tree reordering did not always produce better results than those generated without tree reordering. The genetic algorithms may be improved by applying better crossover and mutation operations. A longer run time for the genetic algorithms may also produce better results. The algorithms may be refined to maintain better adjacency properties or may be modified depending on an application’s properties.

References

[AJU77] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, 1977.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8), 1986.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 1982.

[ChD88] C-H. Chi and H. G. Dietz. Register allocation for GaAs computer systems. *IEEE Proceedings of the 21st Hawaii International Conference on Systems Sciences, Architecture Track*, 1, January 1988.

[Chi89] Chi-Hung Chi. Compiler-driven cache management using a state level transition model. *Ph.D. Dissertation, Purdue University*, 1989.

[CSS99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9, New York, NY, USA, 1999. ACM Press.

[Die92] H. G. Dietz, "Common Subexpression Induction," Proceedings of the *1992 International Conference on Parallel Processing,* Saint Charles, Illinois, August 1992, vol. II, pp. 174-182.

[Die03] H. G. Dietz, "Programmable Nanocontrollers For Nanodevices", Electrical & Computer Engineering Department, University of Kentucky

[DiK93] H. G. Dietz and G. Krishnamurthy, "Meta-State Conversion," *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 47-56, Saint Charles, Illinois, August 1993.

[SeU70] R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions". *Journal of the ACM, http://doi.acm.org/10.1145/321607.321620*, 17(4), 1970.

Vita

- Date and place of birth

  October 19, 1979. Medak, Andhra Pradesh India.
- Educational institutions attended and degrees already awarded

  Padmasri Dr. B.V. Raju Institute of Technology, affiliated to Jawaharlal Nehru Technological University, Narsapur, Medak, Andhra Pradesh, India.

  Bachelor of Technology, Electrical and Electronics Engineering
- Professional positions held

  Software Engineer

  Firmware Engineer II

  Firmware Engineer

  Associate Embedded Software Engineer
- Scholastic and professional honors

  Padmasri Dr. B. V. Raju Gold Medal for academic excellence, 1997-2001
- Professional publications
  - Shashi Deepa Arcot, Henry Dietz and Sarojini Priyadarshini Rajachidambaram, "**Manipulating MAXLIVE for Spill-Free Register Allocation**", Languages and Compilers for Parallel Computing (LCPC05), October 20, 2005.
  - Henry G. Dietz, Shashi D. Arcot and Sujana Gorantla, "**Much Ado about Almost Nothing: Compilation for Nanocontrollers**", LCPC 2003: The 16th International Workshop on Languages and Compilers for Parallel Computing.
  - Shashi D. Arcot and Henry G. Dietz, "**Programmable Control for Nanofabricated Devices**", 2004 International Workshop on Nanomaterials.
- Typed name of student on final copy

  SHASHI DEEPA ARCOT