Manipulating MAXLIVE For Spill-Free Register Allocation

Shashi Deepa Arcot, Henry Gordon Dietz, and Sarojini Priyadarshini Rajachidambaram

Electrical and Computer Engineering Department, University of Kentucky

Abstract. Many embedded systems use single-chip microcontrollers which have no on-chip RAM. In such a system, the processor registers must hold all live data values. *Nanocontrollers* further reduce the controller circuit complexity so that a nanocontroller can be embedded with each of thousands to millions of sensors, actuators, or other devices on a single chip. This reduction in circuit complexity is accomplished by using a bit-serial multiplexor-based SIMD architecture with just tens of one-bit local registers. These registers not only must hold all declared and temporary values, but also are used to hold program state information in support of MIMD programmability. Implementing word-level operations using bit-serial multiplexor operations often yields huge basic blocks with very complex DAGs, apparently requiring even more registers. Spilling is not possible, so code that needs too many registers simply cannot be run.
This paper explores new compilation methods, including Genetic Algorithms (GAs) and a new adaptation of Sethi-Ullman numbering, to aggressively restructure the code and allocate registers so that the number of nanocontroller registers used does not exceed the number available. The approach also is shown to be adaptable to solve the less demanding problem of avoiding register spills for microcontrollers or general-purpose processors.

1 Introduction

The problem of efficiently allocating registers for temporary values is an old problem, but also is a topic of ongoing research. In large part, the importance of register allocation has been increasing because:

- Although both logic and memory speeds have been exponentially improving, the exponents are different. Main memory was once faster than processor logic for simple operations such as integer addition, but modern processors can perform hundreds to thousands of integer additions in the time taken to make one random address access to main memory.
- Registers play a key role in implementing instruction-level parallelism (ILP). Superscalar (multiple issue) execution logic may require many operands each clock cycle. As compared to multi-port caches and main memory interfaces, it is relatively straightforward to construct multi-port register files. Registers also facilitate pipelined execution.
- A variety of automatic coding mechanisms tend to generate much larger basic blocks with more complex dependence patterns than are commonly found in hand-written code. For example, many compilers now use loop unrolling or unraveling; similar code sequences also are generated automatically by tools like ATLAS[15].

While all three of the above increase the importance of register allocation, the first two primarily increase the benefit in using a good allocation, while the third essentially implements a qualitative change in the register allocation problem itself. In the general case, optimal allocation of registers is known to require more than polynomial time, but it is only with the common use of huge basic blocks that the theoretical complexity has become a serious practical constraint on basic block algorithms. Thus, register allocation has become critical at the same time that the known optimal solutions have become intractable.

Beyond the needs of conventional computing systems, we have recently become focused on finding ways to bring programmable intelligence to nanofabricated and MEMS devices; these very simple computing elements are called *nanocontrollers*[8]. For the specific problem of allocating registers for nanocontroller programs, the second of the above issues does not apply, but first and third are exceptionally severe. There literally is no main memory in a nanocontroller system; thus, using memory to hold values that could not be allocated to registers is not an option. Further, because nanocontrollers provide only a single type of instruction which operates on one bit at a time, basic blocks often contain thousands of instructions. These basic blocks are not the result of unrolling, but of bit-level logic optimization using the ternary 1-of-2 multiplexor operation. The dependence structure within a block is correspondingly more complex than that

generated by unrolling loops involving traditional binary operations. In summary, nanocontroller register allocation is a much harder problem than conventional register allocation, but a good solution also may be adapted to handle microcontrollers and future generations of conventional processors.

Our goal in this paper is to be able to generate spill-free code for any of a wide class of modern target processor instruction sets covering conventional, microcontroller, and even nanocontroller designs. The techniques developed in this paper assume that we are successful in achieving this goal, i.e., no mechanism is described for handling spills and reloads. However, the techniques are completely general in all other respects. In particular, the techniques work well for *very large* basic blocks using any combination of unary, binary, and ternary operations.

Section 2 reviews some of the traditional approaches and issues involving register allocation. Our first and more conservative approach, which uses a Genetic Algorithm (GA) to reorder instructions, is detailed in Section 3. An extreme, but amazingly effective, approach combining aspects of Sethi-Ullman numbering with a Genetic Algorithm is described in Section 4. Brief conclusions are given in Section 5.

2 Traditional Approaches To Register Allocation

The term "register allocation" is commonly applied to a wide range of slightly different problems involving making efficient use of registers. These problems include minimizing the estimated cost of spill/reload code, allocation of registers across basic block boundaries, and reordering of instructions to improve the register allocation.

2.1 Minimizing Spill/Reload Cost

The majority of research in register allocation centers on the allocation of values to registers and memory-based temporaries so that the memory accesses required have minimal cost. Some of these techniques also can be applied across basic block boundaries. Unfortunately, because nanocontrollers and some microcontrollers literally have no memory, any spill from a register to memory has essentially infinite cost. Thus, these techniques do not apply unless they happen to find an allocation which has nothing allocated to memory temporaries.

Shortest Path Algorithms Although early computers did not have enough registers or compiler technology to make automated register allocation a major concern, by the 1960s various methods were developed, including an approach using a shortest path algorithm[10] for allocating registers within a basic block. The technique involves creating a multi-stage acyclic graph in which the Kth stage corresponds to possible register file states at the Kth programmed reference in the block. Each node represents the state of the register file as the relevant register contents; arcs carry the execution-time costs for transitioning from one register file state to the next. Thus, selecting the shortest path through the graph yields the time-optimal sequence of register assignments using expected execution times for each potential memory spill/reload operation.

This shortest-path technique was extended in the late 1980s for compiler management of both registers and cache[4,5]. The additional concept of cut-point states enables processing of very large basic blocks and even code regions containing arbitrary control flow. A *cut point* is essentially a stage in which the register file contents are specified, thus allowing the shortest path problem to be decomposed into independent subproblems before and after the cut point. In some cases, the optimal set of register contents at a particular reference is obvious, in which case a cut point occurs naturally without affecting optimality of the solution. Various techniques for artificially inducing cut points allow the shortest-path problems to be bounded to a specified maximum size with little or no reduction in the quality of the resulting allocation.

Graph Coloring In the early 1980s, it became popular to view register allocation as coloring of an interference graph, and this general approach still is used in many production compilers. Each node in the graph represents either a unique value or a variable; arcs are drawn between nodes that have overlapping lifetimes. Such an interference graph easily can be constructed for either a single basic block or code containing arbitrary control flow.

Optimal graph coloring is hard, so various heuristics have been used. Chaitin's node-removal algorithm[3] is perhaps best known, although even a simple random walk typically outperforms it[4]. Many variations now exist, including approaches using GAs [9]. However, the strength of the coloring approach also is its weakness: the actual number of spill/reload events depends on the precise reference sequence, not just (potential) overlap of lifetimes. Thus, costs are approximate.

Other Spill/Reload Cost Minimization Algorithms Work in register allocation, primarily centered on spill/reload minimization, continues. For example, in 2003 a paper was published[11] showing that Belady's MIN algorithm for page replacement performed very well in reducing spill/reload costs for large basic blocks. This is not surprising in that MIN actually is a degenerate case of the shortest path formulation in which cost per spill/reload is assumed to be a constant. This assumption implies that minimizing the number of spill/reload events is equivalent to minimizing the execution time. MIN accomplishes this very efficiently by always spilling the value which will not be accessed again for the largest number of instructions.

2.2 Other Register Allocation Algorithms

As mentioned earlier, for nanocontrollers and some microcontrollers the sole concern is finding an allocation which does not exceed the number of registers available. Any of the above techniques can be used, as can techniques that use GAs or Genetic Programming (GP) to optimize the compiler optimization control structure in the hope of obtaining better allocations [6], but only in the degenerate case where no spills are needed.

Indeed, finding a spill-free register allocation is trivial provided that MAXLIVE, the maximum number of values (or variables) that must temporally coexist, never exceeds the number of registers available. Because nanocontrollers and some microcontrollers do not have ILP issues, such as pipeline schedule interlock constraints, any spill-free allocation will result in the minimum possible execution time.

No register allocation scheme can achieve a spill-free allocation with fewer than MAXLIVE registers. Thus, register allocation for nanocontrollers is primarily a matter of reducing MAXLIVE so that the number of registers is not exceeded. Only reordering of the instruction sequence or changing the computation can change MAXLIVE.

2.3 Sethi-Ullman Numbering

One of the most efficient general-purpose register allocation schemes is commonly known as Sethi-Ullman Numbering (henceforth referred to as SUN)[14]. This algorithm, published in 1970, not only determines how to allocate registers, but also how to order evaluation of an expression so that the number of registers required and the number of instructions used for the computation both are provably minimal.

The assumptions made by SUN are straightforward and even today, 35 years after the algorithm was first published, these assumptions can be met by most computer designs. There are assumed to be $N>=1$ general-purpose registers, any of which may interchangeably be used as a source or destination in an operation. The region of the program considered by the algorithm is a single arithmetic expression involving binary operations. The relationships between these operations are expressed as a binary tree that links each binary operation to the two operations that provide its operand values. Leaf nodes in the tree represent initial values of variables and constants.

The SUN algorithm proceeds in two distinct phases. First, each node is labeled with a number, according to a set of rules, such that the label corresponds to the minimal number of registers required to evaluate the subtree rooted at that point without any stores (i.e., without register spill/reload). These labels are then used to order node evaluation, allocate registers, and emit instructions.

A bottom-up walk of the binary tree is used to assign to each node *n* the label *L(n)*. The algorithm given by Sethi and Ullman[14] uses the following two rules to assign labels to nodes:

1. If *n* is a leaf and a left descendant, *L(n)* = 1.
   If it is a right descendant , *L(n)* = 0;
2. If *n* has descendants with labels l_1 and l_2,

(a) If $l_1 != l_2$, $L(n) = \max(l_1, l_2)$;
(b) If $l_1 == l_2$, $L(n) = l_1 + 1$

Rule 1 reflects the additional assumption that the binary instructions support a register-memory instruction model in which the right descendant can be accessed directly from memory, provided that the left descendant is loaded into a register. In other words, an instruction can be of the form *register = operation(register, memory)*, absorbing the fetch of the right operand into the parent instruction.

Most current processor designs either have RISC-like instruction sets without support for memory operands or, despite having instruction set support for memory operands, have recommended coding practices that avoid using memory operands. Such machines are trivially accommodated by removing the distinction between left and right descendants in rule 1; a leaf node *n* is always given $L(n) = 1$ reflecting the fact that any memory operand must be loaded into a register before use. Another minor change since the original SUN was developed is that many compiler systems, including GCC and various compilers targeting microcontrollers with special-purpose registers, now support pre-allocation of variables to specific registers; a leaf node *n* which refers to a pre-allocated register's value is always given $L(n) = 0$ because it is not necessary to move the value to a different register in order to operate on it.

Rule 2 reflects use of register-register operations for internal nodes of the tree and thus needs no modification.

After the tree is generated and the nodes labeled, the algorithm proceeds as a recursive walk starting at the root node, selecting an evaluation order for the descendants of each node in which the operation with the 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 actually the maximum number of live values (MAXLIVE) 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 each node a register. If that number is exceeded, then SUN provides a straightforward way in which values can be selected to be spilled from registers into memory and reloaded when necessary.

This entire procedure visits each node at most a constant number of times, thus yielding $O(m)$ complexity for scheduling and allocating registers for *m* operations. Despite this speed, the evaluation order and allocation both are optimal in that, given the assumptions made, SUN uses the *fewest instructions* to accomplish the tree's computation. At the time SUN was developed, using the fewest instructions closely corresponded to minimizing execution time, minimizing spills, and minimizing the number of registers used.

Given the simplicity of the algorithm and optimality of the results, it is rather surprising that SUN is not in common use in modern compilers. There are several reasons why SUN is not used, foremost being the fact that SUN cannot be directly used to analyze a code region that is more complex than a tree computing a single value. Common Subexpression Elimination (CSE) greatly reduces the number of instructions that need to be executed, but generates Directed Acyclic Graphs (DAGs) that are incompatible with the original SUN algorithm and a multitude of attempts to extend SUN to handle DAGs have failed to produce an algorithm that is both fast and effective[1]. Given how slow modern computers are to access memory, perhaps it would be better to favor use of SUN over CSE, but that is not the path that compilers have generally taken.

For nanocontrollers and some microcontrollers, the primary targets in this paper, the problem is qualitatively different: even a single spill renders a program unusable if there is no place to spill to. Thus, minimizing the number of instructions only is relevant if the code is spill free. Put another way, even increasing the number of instructions to be executed is highly desirable if it makes the difference between being spill-free and being unusable.

3 Genetic Algorithm For Reordering To Minimize MAXLIVE

Given that reordering the instruction sequence can significantly change MAXLIVE, it seems appropriate to investigate methods that can reasonably efficiently find a good instruction order. Even with good pruning, it is not practical to use exhaustive search for reordering basic blocks containing thousands of instructions. However, simulated evolutionary processes are very effective for many conceptually similar problems, so we created a Genetic Algorithm (GA) for reordering.

Algorithm 1 Steady-State Island GA For Scheduling

Repeat the following until the allotted time or number of trials has elapsed:

1. If the population is not yet full, create a new valid, but randomly-ordered, instruction schedule; goto step 5
2. Pick a number of population members at random and identify the two selected members with the worst and best metrics (a form of tournament selection); an island model may be enforced at this stage by biasing selections to stay within the same static subdivision of the population
3. If random choice selects mutation or if the two schedules selected are duplicates, perform mutation by replacing the poorest-metric selected member with a new schedule created by mutation of the other selected member; goto step 5
4. By default perform crossover by picking an additional population member at random, sorting the three selected members by metric value, and replacing the poorest-metric one with the crossover product of the other two
5. Evaluate the metric for the newly-created population member
6. Determine if the newly-created population member is a new best and mark it accordingly; it is the new best if it is the only member of the population or if a symmetric "better than" comparison function finds its metric to be better than that of the previous best schedule

3.1 Structure Of The GA

The use of a GA to generate code is commonly referred to as Genetic Programming (GP)[13], however, neither the data structures standardly used with GP nor with traditional GA systems is efficient in solving the instruction rescheduling problem. Despite that, the overall structure of the GA used for rescheduling to minimize MAXLIVE, as shown in Algorithm 1, is relatively conventional. An island model is used in order to allow subdivisions of the population to converge to different solutions in relative isolation, thus making the system somewhat more robust. A non-generational steady-state formulation is used primarily to simplify the coding and reduce execution overhead.

Fundamentally, the problem in making the GA efficient is one of maintaining good adjacency properties through mutation and crossover operations; a new schedule should have many properties in common with its parent(s). In the particular case of instruction scheduling, it also is important to consider only valid schedules, e.g., only schedules in which no instruction is scheduled before an instruction that produces one of its inputs. Even using simplifications such and earliest and latest slot markings for instructions, checking validity of a schedule is relatively expensive. Discovering that a schedule is not valid also wastes the effort of creating and checking that schedule. Thus, the preferred solution is to generate only valid schedules.

This is done by using an unusual geneome representation which we have recently used for several types of scheduling GAs: rather than representing an instruction schedule directly, a schedule is represented by giving each instruction an integer “scheduling priority.” The schedule is generated using these priorities to break ties in an otherwise conventional list scheduling procedure. The schedule is created by starting with the first instruction slot and working toward the last, at each slot updating the set of schedulable instructions and then inserting the highest priority schedulable instruction in that slot. Clearly, only valid schedules are produced in this way. Further, most adjacency properties are inherited from parent(s) even though the actual schedules may differ in what appear to be complex ways; changes in priorities may rearrange, spread, insert, or delete subsequences of instructions, but before/after relationships between instructions with priorities that were not changed by mutation or crossover are most often preserved. It also is trivial to compute a MAXLIVE-based metric while generating the schedule.

The mutation and crossover operations are straightforward. Mutation replaces some priorities with random values, whereas crossover mixes priorities from two parents. Interestingly, as a schedule is being assembled for evaluation, it is easy to tag each instruction with the number of live values at its position in the schedule, and hence to know which instructions are involved in subsequences requiring MAXLIVE registers. Thus, we can bias the mutation and crossover operations to change priorities for instructions in those regions, significantly improving the speed of convergence.

3.2 Experimental Procedure

In order to determine just how well the reordering GA works, we constructed a test framework which we have used for all the data presented in this paper. The framework consists of:

– A simple program to generate pseudo-random BitC programs containing a single basic block each. BitC is a simple C dialect designed from programming nanocontrollers[8]; it differs from C primarily in that it allows bit precisions to be specified for each variable and incorporates some additional operators, such as binary minimum and maximum (`?<` and `?>`).
– The base BitC compiler which we earlier developed for our research in nanocontrollers, `bitcc`. This compiler converts each variable-precision word-level operation into a multitude of single-bit operations implemented using the only operation provided by nanocontrollers, the ITE (If-The-Else) 1-of-2 multiplexor function. The operations are then optimized by a variant of BDD (Binary Decision Diagram) logic minimization methods[2,12], yielding better code than simple bit-slice formulations would, but with very complex DAG structures. In the `bitcc` output used for the current study, storage of final values into registers is done by separate explicit store operations.
– An ITE+store to SITE (Store-If-Then-Else) converter constructed specially for this research. This program removes the explicit stores, combining them with ITEs in an optimal way. Thus, sets of operations like `temp=(i?t:e); s=temp;` are converted into `s=(i?t:e);`. The SITE-only DAG, which incorporates a reference sequential order, is then coded as a set of C data structures and output to `dag.h`. This "pre-cooked" set of data structures makes it much easier to perform register allocation experiments by avoiding the need to integrate the algorithm under test with the rest of the compiler.
– The GA reordering code described above. Thanks to including `dag.h`, this code can be modified and re-run without the overhead of BitC compilation; the entire program is just over 300 lines of C code.
– The SUN-based GA described in Section 4. Again, thanks to including `dag.h`, this entire program is short: just under 600 lines of C code.
– A variety of shell scripts and filters to run tests and collect data. Relatively simple cases occur very often in randomly-generated code, for example, when a later store into a variable overwrites the value stored by a more complex computation very little code results. Thus, our methodology includes a filtering step that removes all cases with MAXLIVE less than 3. Additionally, filters are applied to remove statistically redundant cases.

Using this framework, we collected data on millions of test cases. Our scripts allow large numbers of test cases to be executed serially or in parallel on cluster supercomputers.

The results presented in scatter plots in this paper were computed using KASY0 (Kentucky ASYmmetric Zero), a 128-node 2GHz Athlon XP system. All the GAs were given the same fast-running parameters: population size of 50, subdivided among 4 islands, with crossover 3 times more likely than mutation, and a limit of evaluating only 1,000 individuals. There are 32,912 cases in the filtered test case set presented.

3.3 Results

At the outset, in early 2004, we had hoped that reordering instructions would be sufficient to dramatically reduce MAXLIVE, but experimental results are mixed.

For relatively modest basic block sizes, such as those commonly arising from hand-written code in languages like C for targets like IA32, the GA reordering does well. However, ternary instructions and larger basic blocks tend to yield not just larger, but also more complex DAG structures. Our preliminary tests showed that, for the large ternary instruction basic blocks common in nanocontroller code, GA reordering reduced MAXLIVE significantly in absolute terms, but not enough to make a qualitative difference for our nanocontroller compilation problem. These (unpublished) early observations are echoed in the more extensive data presented here.

The GA reordering of instructions does not change the total number of instructions which must be executed (assuming no register spill/reload operations are needed), nor does it alter the underlying DAG structure. Thus, the only relevant issue is the reduction in MAXLIVE, which is shown in the scatter-plot of Figure 1. Note that both axes in this graph are logarithmically scaled. As observed in preliminary experiments, although MAXLIVE is reduced more in absolute terms for the larger cases, the relative reduction for relatively small cases is significantly larger than for larger cases. The average reduction over all 32,912 cases is approximately 18%. Thus, while these results clearly confirm that GA reordering is well worth applying, it alone is not sufficient for nanocontroller targets – which are expected to provide only about 64 registers.

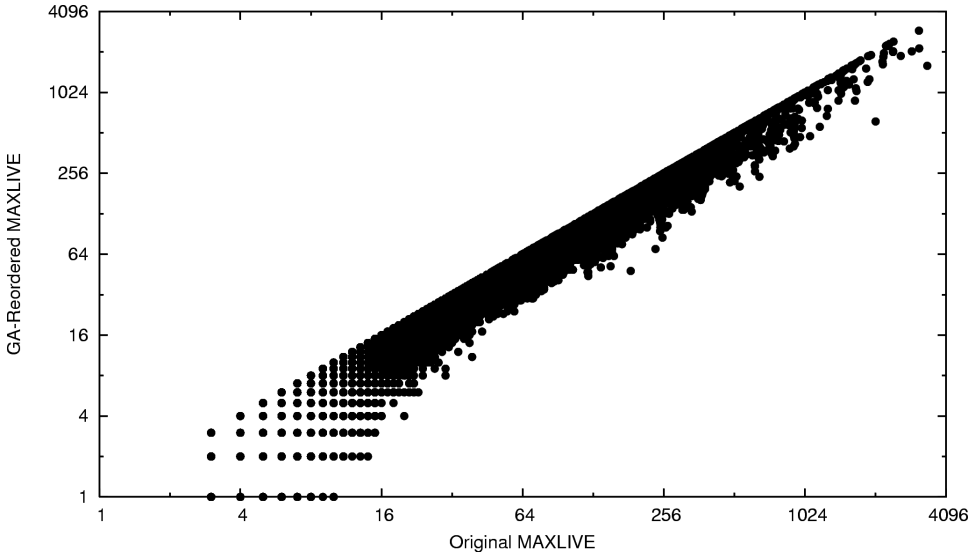

Fig. 1. GA-Reordered Vs. Original MAXLIVE

4 SUN With GA-Reenabling Of CSEs

Given that even GA reordering of instructions is not sufficient to make big blocks spill free, it is necessary to consider techniques that trade execution of more instructions for a more dramatic reduction in MAXLIVE.

The approach is based on the SUN algorithm, but makes considerable extensions to it. The first extension is the generalization of SUN to manage up to three operands per instruction. This modification is required because the SUN algorithm as originally presented assumes each single-instruction operation takes precisely two source operands, yet the only instruction supported by current nanocontroller designs takes three source operands and different operand counts may be useful for other types of specialized processors.

As suggested earlier, the lack of register-memory instructions requires only a minor adjustment to the SUN algorithm, but three other issues are more difficult to resolve. There have been many attempts to extend SUN to handle optimal register allocation and instruction scheduling for DAGS. Although, under certain restricted conditions, DAGs can be handled using a modified SUN algorithm, the optimality of the solution is a casualty in every reasonably efficient scheme. 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 DAG for which special-case extensions of SUN can be applied. Our solution is to convert the DAG to a tree by logically replicating every common subexpression in every place from which it is referenced. This solution may seem extreme, but the DAG generally has an inherently higher MAXLIVE than a tree; given the extreme pressure to fit in a limited register file, it is natural to focus first on minimizing MAXLIVE and only secondarily to attempt to retrieve some of the benefits of common subexpression elimination.

4.1 Generalization Of SUN Labeling For Ternary Instructions

The labeling method used in the original SUN algorithm is focused on binary operations: instructions with two input operands. Unary operations are trivially labeled using the rule that any operation node *n* with only one input operand is labeled with $L(n) = 1$. It is not trivial to extend SUN labeling to three or more input operands. However, digital nanocontrollers as currently proposed have an instruction set consisting of only a single instruction which happens to take three input operands. Three-input operations, generally involving multiplexor-like functionality used to simulate enable masking, also have become common in multimedia instruction set extensions to many modern processors[7].

The labeling of three-input operation trees is significantly more complex than that of two-input operation trees because the number of possible relationships between subtree labels grows exponentially as the number of inputs per operator increases. To each node *n*, the label $L(n)$ is assigned as:

1. If *n* is a leaf, $L(n) = 0$;

2. If *n* has descendants with labels l_1, l_2, and l_3 sorted into order such that $l_1 >= l_2 >= l_3$,
   (a) If $l_1 > l_2 > l_3$, $L(n) = l_1$;
   (b) If $l_1 > l_2 == l_3 == 0$, $L(n) = l_1$;
   (c) If $l_1 > l_2 == l_3 != 0$ and $l_1 - l_2 == 1$, $L(n) = l_1 + 1$;
   (d) If $l_1 > l_2 == l_3 != 0$ and $l_1 - l_2 > 1$, $L(n) = l_1$;
   (e) If $l_1 == l_2 > l_3$, $L(n) = l_1 + 1$;
   (f) If $l_1 == l_2 == l_3 != 0$, $L(n) = l_1 + 2$;
   (g) If $l_1 == l_2 == l_3 == 0$, $L(n) = 1$;

Rule 1 reflects the now-common simplifying fact that modern processors avoid using memory operands directly. For example, leaf nanocontroller operations always can be labeled with L(n) = 0 because there literally is no way for an instruction to reference data other than making a register reference. Constants are referenced from pre-allocated registers; given bit-wide data paths and operations, only the constants 0 and 1 are possible, so hardwiring just two pre-allocated registers suffices. Nanocontrollers have only registers in which to store data, so in fact all user-defined variables become preallocated registers. Nanocontrollers even perform input/output (I/O) operations using pre-allocated registers that are really I/O channels; for example, register 6 might be a “global OR” output signal and register 7 might be an analog zero-crossing detector input. Data can be directly used from a pre-allocated register identically to how it would be used from any other register; no load instruction in needed (or even exists for nanocontrollers).

Rule 2 reflects register needs for non-leaf nodes. As complex as this rule is, the complexity is significantly reduced by the fact that it is expressed in terms of the labels of the three input subtrees in an order that is sorted by label. Thus, l_1, l_2, and l_3 are the descendant labels in decreasing label order, not subtree position order. The complexity of this rule is still high primarily because equal labels and labels of 0 are both special cases. However, in practice, the complexity of the rule has little impact on the feasibility of the technique. It also is useful to note that the ternary node case also handles both binary and unary node labellings by allowing the missing descendants to be treated as if they had 0 labels.

4.2 Tree Generation

At the time the SUN algorithm was proposed, it was quite natural to use trees as the intermediate form. However, coding styles have significantly changed, so that various compiler optimizations yielding DAGs are now essentially mandatory. For nanocontroller programs, these DAGs are particularly large and complex thanks to treatment of each bit position separately and target hardware support for only one type of instruction (which corresponds to a 1-of-2 multiplexor).

As stated earlier, nanocontroller programs generate optimized DAGs which are large and complex. Each SITE that is generated is a node in the DAG. The root node(s) of every DAG corresponds to a SITE that is a final store into a variable. All the interior nodes correspond to the temporary SITEs which represent the ITE operations. By convention, our tools number these starting at 64, the default number of physical nanocontroller registers available. The leaf nodes are the ITEs 0 and 1 or the ITEs that correspond to the initially defined user-variables – nodes numbered less than 64. Trees are generated by conceptually converting all the DAGs to trees in such a way that each node is replicated at every point that it is referenced.

To demonstrate the treatment of a DAG as a tree, consider the following simple example:

```
64: 2 0 1
65: 4 64 2
66: 3 0 1
67: 3 2 64
68: 4 67 66
```

The above 5-ITE basic block not only provides a default sequential order, but also naturally embeds the perhaps surprisingly complex DAG shown in Figure 2. Ternary nodes tend to yield more complex DAGs than do binary nodes.

Although the SUN algorithms cannot operate on a DAG, it is easy to treat the DAG as a tree. Logically, the transformation is simply that, whenever a node has more than one exit arc, the node is replicated to make

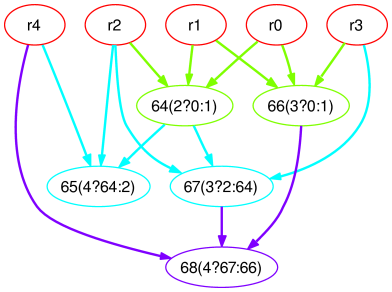

Fig. 2. Simple ITE DAG

one copy per exit arc. As a node is thus replicated, any entry arcs must also be replicated to point at the copies. This in turn makes the nodes behind those entry arcs have multiple exit arcs, thus requiring them to be replicated in the same fashion. The result of this transformation is shown in Figure 3.

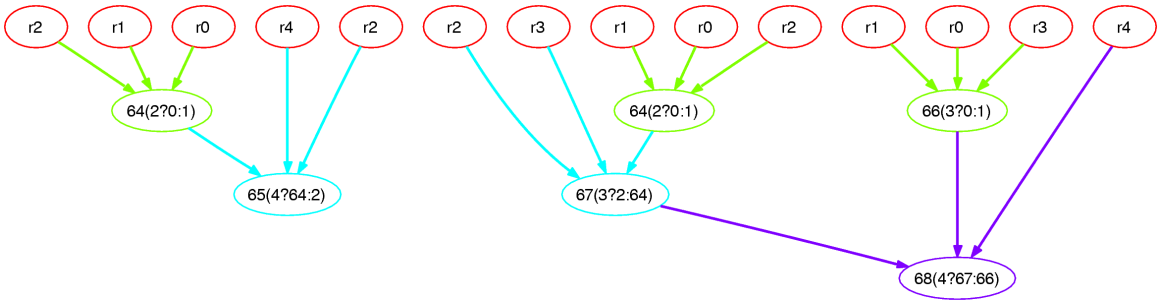

Fig. 3. Trees Derived From Simple ITE DAG

A subtle point in this transformation is the fact that a single DAG becomes multiple trees. Even if the original DAG had unconnected components, the default sequential order (as listed above) can yield a default execution order. For our purposes, the SUN algorithm will provide the order within each tree, but ordering across trees must be provided in another way. The solution used in this paper is to order the tree walks in the same order as the nodes without exit arcs were originally ordered. Thus, in Figure 3, the tree ending in 65 (right) would be evaluated before the one ending in 68 (left).

Of course, the transformation to create a tree does not merely enable SUN analysis, but also provides a key relationship between nodes that are the roots of common subexpressions in the DAG. We can use the rules of our modified SUN to label tree nodes for walking, thus implying a walk order, but then not actually duplicate the common subexpression nodes. This is the core idea behind the SUN-based GA: to use a Genetic Algorithm (GA) to selectively re-enable CSE (Common Subexpression Elimination); where MAXLIVE will not be too adversely affected using the walk order determined using the tree, do not replicate the common subexpression node.

4.3 GA Optimization Of Subexpression Instantiation

It should not be surprising that the basic steady-state island GA structure of Algorithm 1 also serves well for the SUN-based GA. The details are surprisingly straightforward, as outlined in Algorithm 2.

Whereas the GA-reordering algorithm described in Section 3 required a fairly complex data structure, our SUN-based GA for selective reinstantiation of CSEs can effectively use a very conventional bit-sequence geneome. Each geneome is a bit vector with one bit for each potential CSE; a 1 means instantiate (i.e., the CSE is enabled), a 0 means duplicate to make a tree.

To evaluate the merit of a geneome, the DAG is recursively walked as a sequence of trees (as per Section 3). The walk uses the labels and ordering of operand evaluation created by treating the DAG as a tree

Algorithm 2 SUN-Based GA Procedure Overview

1. Use the tree interpretation (Section 4.2) of the DAG to label nodes as described in Section 4.1. Note that interpreting the DAG as a tree does not require literally duplicating nodes; no node copies are made in our coding. The labeling can even take advantage of the fact that CSEs need only be traversed once to be labeled, because additional traversals would yield the same labels.
2. Apply the steady-state island GA (Algorithm 1),with the following adjustments:
   (a) The initial population is loaded with both the tree (no CSEs instantiated) and original DAG (all CSEs instantiated) as members in addition to random members.
   (b) As the search progresses, the evaluation of any population member can be truncated when its value of MAXLIVE reaches a "terrible" level that can be specified as input to the GA and also can be dynamically updated as better MAXLIVE values are encountered in the search.

and applying the rules in Section 4.1. As each node is visited, it is allocated a register if needed. Nodes representing enabled CSEs are walked only the first time they are encountered. After the value of a non-CSE node has been used, the register allocated to it is freed. The register allocated to an enabled CSE node is freed only after no reference to that CSE remains, which is determined by decrementing a reference count associated with that node. The value of MAXLIVE and number of instructions that would be generated by the walk are both tracked during the evaluation; as noted in Algorithm 2, the recursive walk can be aborted early if MAXLIVE becomes too large. The metric favors generating fewer instructions once the MAXLIVE constraint has been met.

The mutation and crossover operations are very standard GA bit-geneome operations. The only notable difference is that random choices are made for each bit position in crossover, rather than using the even more common subsequence interchange. The randomly generated (initial) population members are created using a two-step process that first selects a random target "loading" and then randomly turns on bit positions to achieve that loading; this yields a better coverage of the full range of CSE enable densities.

Overall, the SUN-based GA is a very standard GA that has an unusual merit evaluation process.

4.4 Results

Testing the SUN-based GA for selectively enabling common subexpression elimination immediately revealed that the concept of allowing some redundant evaluation was able to dramatically reduce MAXLIVE. In fact, the reduction possible for large blocks is nothing short of shocking, with nearly every nanocontroller test case collapsing to a form using approximately a dozen temporary registers despite initially having a MAXLIVE of hundreds or even thousands.

In order to expose the general relationship between enabling CSEs and increasing MAXLIVE, a series of experiments were conducted using our SUN-based GA to optimize a moderately complex nanocontroller basic block for various target MAXLIVE values. This basic block, with all possible common subexpressions eliminated, consists of 3,041 ternary SITE instructions and yields a MAXLIVE of 561 in its default ordering. In this particular case, our GA reordering the instructions is able to reduce MAXLIVE only slightly, to 553. However, disabling all CSEs results in a pure tree which, using our modified SUN algorithm requires only 12 registers. Unfortunately, the pure tree contains 23,819 SITEs – nearly 8 times as many instructions.

Figure 4 shows how the number of enabled CSEs varies with the MAXLIVE target using our SUN-based GA. All of the CSE counts plotted are for the coding yielding the lowest number of SITEs for the given MAXLIVE target. Surprisingly, the SUN-based GA was able to achieve a MAXLIVE of 12 with 662 CSEs enabled. However, the impact of enabling these 662 CSEs on reducing the number of SITEs is minimal; because some CSEs are nested and the subtree sizes saved by enabling a CSE vary widely, the relationship between the number of CSEs enabled and the total number of SITEs remaining is not direct.

Figure 5 shows how the total number of SITEs varies with the MAXLIVE target for the same test case used in Figure 4. Note that in both figures, MAXLIVE is plotted on the X axis using a log scale. Clearly, although large reductions in MAXLIVE are possible, they come at a high price in additional instructions to be executed. The decrease in MAXLIVE is approximately linear with the increase in SITEs. However, the slope is favorable; as the number of additional instructions increases by nearly an order of magnitude, close to two orders of magnitude reduction in MAXLIVE is realized.

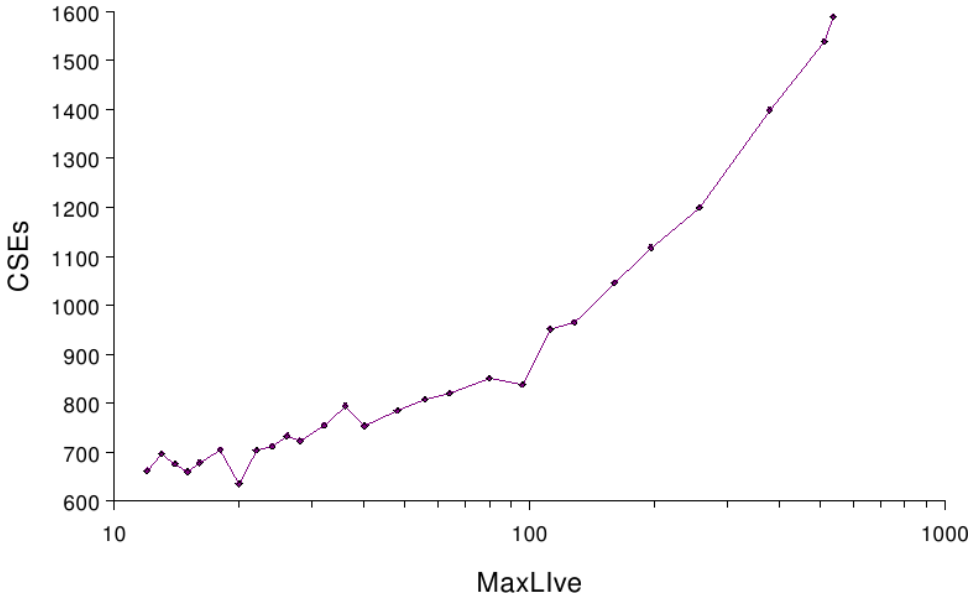

Fig. 4. Enabled CSEs Vs. MAXLIVE For A Nanocontroller Basic Block

The search space is sufficiently large so that exhaustive evaluation of any but the smallest examples is impractical; ignoring the ordering problem, any problem with k potential CSEs has 2^k different code structures to evaluate. For basic blocks of nanocontroller code, k commonly exceeds 1,000 – as it does in this example. Thus, we do not have known optimal solutions for typical problems and cannot make specific claims about the absolute quality of the SUN-based GA results. For Figures 4 and 5, the search was constrained to take approximately one minute to optimize for each target MAXLIVE (running compiled C code on a 1.4GHz Athlon XP system under Linux), and this restriction has no doubt contributed to the noise level visible in the curves for this one test case.

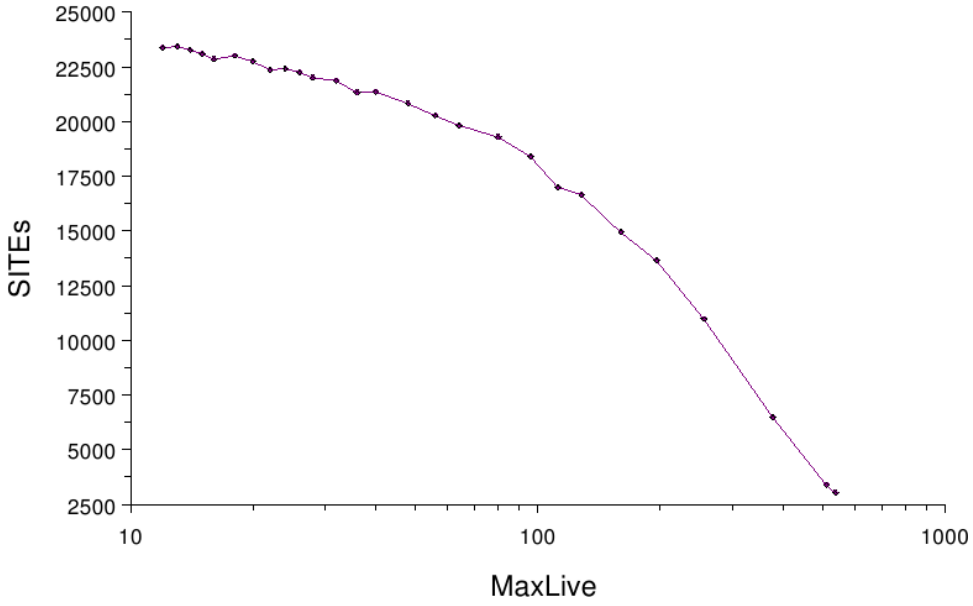

Fig. 5. Number Of SITEs Vs. MAXLIVE For A Nanocontroller Basic Block

In addition to the detailed study of how a specific DAG's processing changes with different target values for MAXLIVE, it is useful to examine the statistical behavior of the algorithm over a large set of cases. For

this purpose, we used the exact same cases that we employed to evaluate the GA for reordering instructions (Section 3.3). This enables direct comparison of the two approaches, as well as statistical evaluation of each independently.

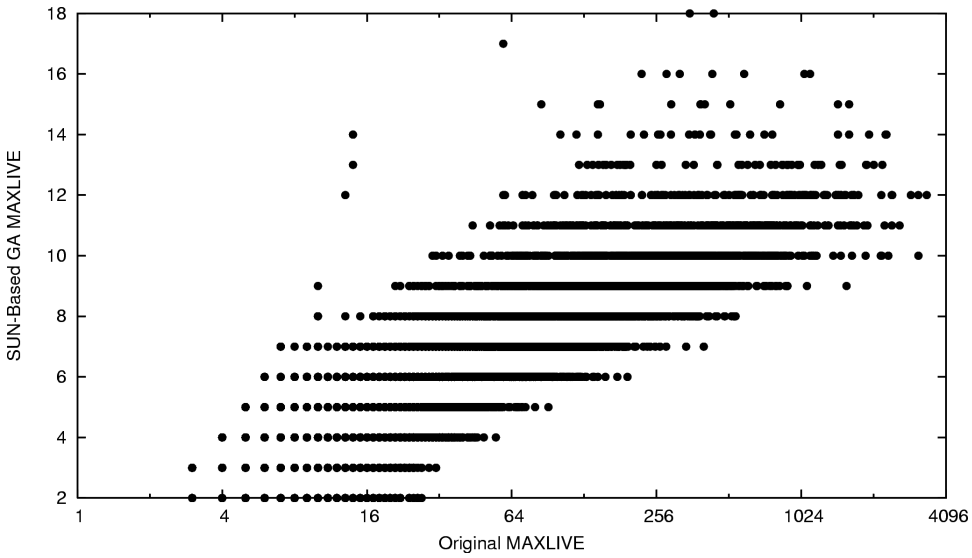

Fig. 6. SUN-Based GA Vs. Original MAXLIVE

Perhaps the most important statistic is how well MAXLIVE can be reduced by the SUN-based GA. Figure 9 shows that the performance in this respect is nothing short of amazing; none of the 32,912 test cases needed more than 18 registers – well within our nominal nanocontroller goal of fitting within 64 registers. Note the logarithmic scale in the X axis of this graph. Even a DAG having a default-order MAXLIVE of 3,409 still fit in 18 registers – more precisely, that case fit in just 12 registers!

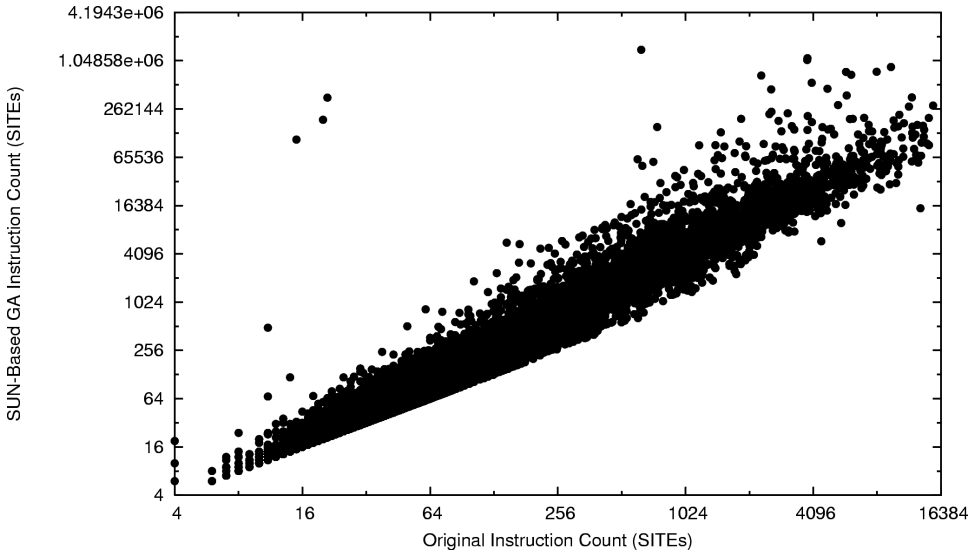

Fig. 7. SUN-Based GA Vs. Original Instruction Counts (SITEs)

Of course, there has to be a catch, and there is. As Figure 9 clearly shows, making MAXLIVE as small as possible often requires executing many more instructions than the original DAG would have required. Note that both axes in this graph are logarithmic, but the largest original block had 15,309 instructions (SITEs) while the largest produced by SUN-based GA had 1,431,548. On average, there was a factor of

8X expansion in code size to obtain the lowest possible MAXLIVE. As dramatic as this tradeoff is, such a code size expansion can be acceptable if it is the difference between being able to use the code and not being able to; even on desktop processors, the penalty for accessing main memory may be high enough to occasionally warrant executing 8X more instructions. Further, recall from Figure 5 that the SUN-based GA is able to efficiently target a specific MAXLIVE target, so it is not necessary to suffer code expansion beyond that needed to reach the target MAXLIVE value.

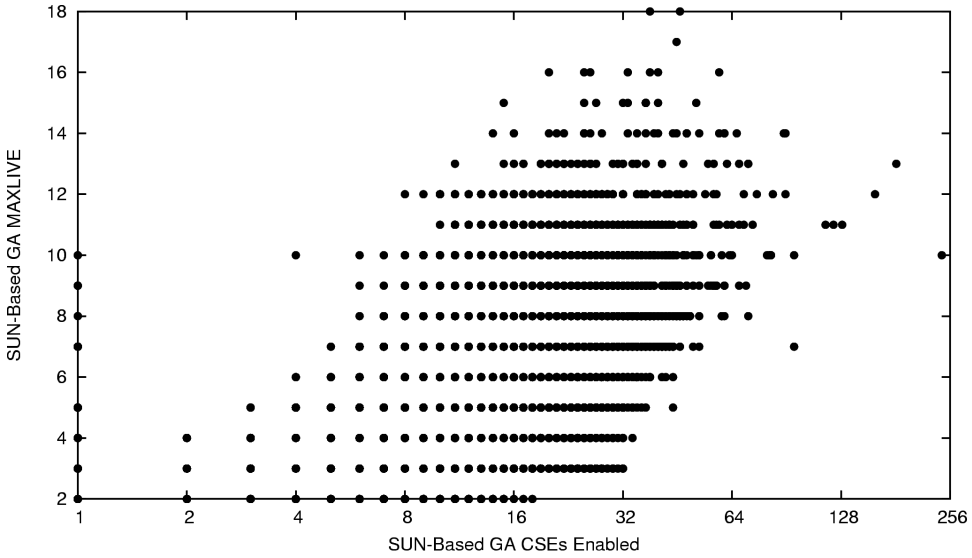

Fig. 8. SUN-Based GA MAXLIVE Vs. CSEs Enabled

Given that the SUN-based GA approach selectively enables CSEs, one might expect that the number of CSEs enabled is essentially zero in order to achieve the minimum MAXLIVE value, but Figure 8 shows that is not the case. A modest reduction in the number of instructions generated is generally possible, without adversely affecting MAXLIVE, by carefully selecting to enable some CSEs.

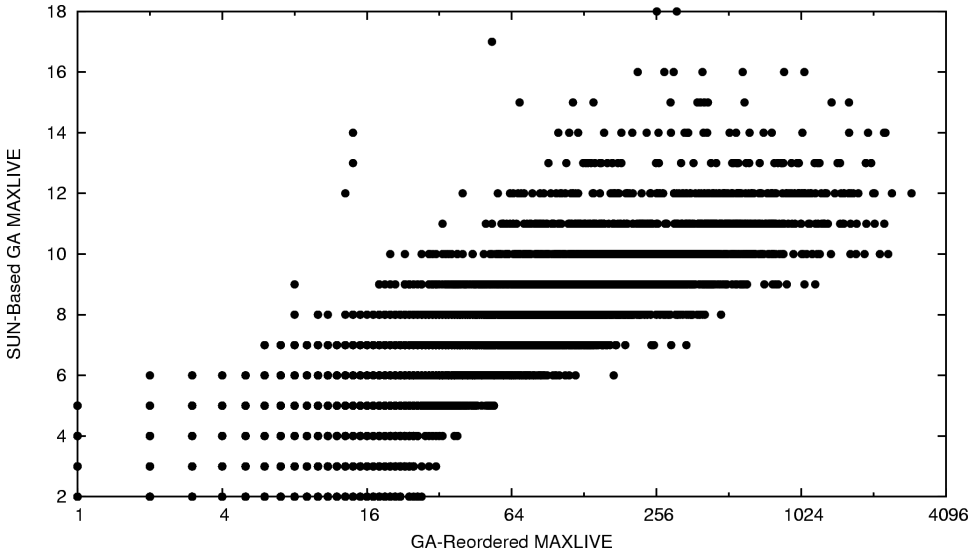

Fig. 9. SUN-Based GA Vs. GA-Reordered MAXLIVE

5 Conclusion

This paper has presented two very aggressive methods for attempting to force an extremely complex block to meet a very small MAXLIVE constraint. One technique, GA reordering, clearly works well and should be widely applied; there is no major penalty. The other technique, SUN-based GA, offers amazing reductions in MAXLIVE, but at the expense of significant code expansion. Figure 9 shows that the SUN-based GA is able to handle extremely complex blocks exponentially better than GA reordering.

If the goal is simply to be spill free, the lowest-cost method that results in a viable MAXLIVE should be used. Often, GA reordering will suffice. When it does not, the SUN-based GA should be used with an explicit cut-off value equal to the number of registers available. Adapting these methods to achieve goals more complex than just freedom from spills, such as simultaneously optimizing pipeline performance or minimizing power consumption, is future work.

References

1. A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, 1977.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8), 1986.
3. G. J. Chaitin. Register allocation & spilling via graph coloring. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 1982.
4. 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.
5. Chi-Hung Chi. Compiler-driven cache management using a state level transition model. *Ph.D. Dissertation, Purdue University*, 1989.
6. 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.
7. H. G. Dietz and R. J. Fisher. Compiling for simd within a register. *Languages and Compilers for Parallel Computing, edited by S. Chatterjee, J,. F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr, and P-C Yew, Springer-Verlag, New York, New York*, 1999.
8. Henry G. Dietz, Shashi D. Arcot, and Sujana Gorantla. Much ado about almost nothing: Compilation for nanocontrollers. *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, 2958:466–480, January 2004.
9. R. Filho and G. Lorena. A constructive genetic algorithm for graph coloring, 1997.
10. L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *Journal of the ACM (JACM), http://portal.acm.org/citation.cfm?doid=321*, 13, January 1966.
11. David Padua Jia Guo, Maria Jesus Garzaran. The power of belady's algorithm in register allocation for long basic blocks. *Languages and Compilers for Parallel Computing, http://parasol.tamu.edu/lcpc03/informal-proceedings/Papers/35.pdf*, 2003.
12. K. Karplus. Representing boolean functions with if-then-else dags. *Technical Report UCSC-CRL-88-28, University of California at Santa Cruz*, November 1988.
13. John R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.
14. 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.
15. R. Whaley and J. Dongarra. Automatically tuned linear algebra software. *Technical Report UT CS-97-366, University of Tenessee*, 1997.