Register Allocation for Indirect Addressing in Loops

Guido Araujo and Sharad Malik

Relatório Técnico IC–98-11

Abril de 1998

Register Allocation for Indirect Addressing in Loops

Guido Araujo and Sharad Malik

Indirect addressing is by far the most used addressing mode in programs running in embedded CISC architectures. The reason is that it enables fast address computation combined with short instructions, resulting in faster/smaller programs. This paper proposes a solution to the problem of allocating address registers to array references within loops, when using indirect addressing combined with auto-increment. The result is a $O(n^{2.5})$ algorithm, based on the solution of a maximum bipartite matching problem, which minimizes the number of address registers required by a program.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Compilers —*Optimizations*

General Terms: Algorithms

Additional Key Words and Phrases: Code generation, register allocation

1. INTRODUCTION

Embedded programs, like those used in audio/video processing and telecommunications, are playing a crescent role in computing. Due to its performance and code size constraints, many embedded programs are written in assembly, and run in specialized processors and/or commercial CISC machines. The increase in the size of embedded applications has put a lot of pressure towards the development of optimizing compilers for these architectures. Processors that run embedded programs range from commercial CISC machines (e.g. Motorola 68000) to specialized *Digital Signal Processors* (DSPs) (e.g. ADSP 21000), and encompass a considerable share of the processors produced every year.

Register allocation is a well studied problem in compilers. Many of the primordial problems in code generation involved finding good algorithms for register allocation [Bruno and Sethi 1976; Sethi 1975; Aho et al. 1977a]. Code generation for expression trees has a number of $O(n)$ solutions, where n is the number of vertices in the tree. These algorithms are used in code generation for stack machines [Bruno and Sethi 1975], register machines [Sethi and Ullman 1970; Aho and Johnson 1976; Appel and Supowit 1987] and machines with specialized instructions [Aho et al. 1977b]. Global register allocation is an important problem in code generation which has been extensively studied [Chaitin 1982; Briggs et al. 1982; Chow and Hennessy 1990; Callahan and Koblenz 1991; Lal and Appel 1997]. Others have considered the interaction of register allocation and scheduling in code generation for RISC machines [Goodman and Hsu 1988; Bradlee et al. 1991], and interprocedural register allocation [Fisher and Kurlander 1996]. Hitchcock III, C.Y. [1986] studied many of the problems involved in using addressing modes. Horwitz et al. [1966] proposed the first algorithm for optimal allocation of address registers in straight line code. The problem of allocating local variables to the *stack-frame*, using indirect addressing and auto-increment, was first formulated and solved by Bartley [1992] and extended later by Liao et al. [1996].

Address computation constitutes a large fraction of the execution time for most programs. Addressing can account for over 50% of all program bits and 1 out of every 6 instructions for a typical general-purpose program [Hitchcock III, C.Y. 1986]. These numbers are probably higher for the case of an embedded program, given that the majority of its operands are array elements. In order to speed up address computation most embedded/CISC processors offer specialized *addressing modes*. A typical example is the auto-increment (decrement) mode, in which an address register is incremented (decremented) after the memory operation is finished. All commercial DSPs and most CISC processors *Instruction Set Architectures* (ISAs) have indirect modes. Indirect addressing enables in-

structions to be encoded into fewer bits than other addressing modes, resulting in smaller programs. Because of that, indirect addressing is largely favored by embedded programmers and architecture designers. Moreover, memory access is designed in such a way that a single memory reference is enough to access typical data types (e.g. *integer*). As a consequence, incrementing (decrementing) an address register is typically the only operation required to compute the address of the next data in memory. This paper proposes a technique that improves address register allocation in architectures which have indirect addressing modes based on auto-increment (decrement). This technique is not targeted exclusively to embedded processors; it can also be used in any architecture which has indirect addressing, including many CISC processors (e.g. Motorola 68000).

This work is divided as follows. Section 2 describes how array references in loops are compiled into indirect addressing instructions. Section 3 presents a graph based formulation for the address register allocation problem. Section 4 proposes a solution for this problem. Finally, Section 5 evaluates the performance of the algorithm, and Section 6 summarizes the main results of this work.

2. ARRAY REFERENCES IN LOOPS

In order to implement indirect addressing, architectures have hardware units, known as *Address Generation Units* (`AGU`), which enable fast address computation. A typical `AGU` has a number of *Address Registers* (`ARs`) and an `ALU` that performs basic arithmetic operations such as increment/decrement. The `ARs` are used to point to the memory position where the desired data is stored. Other architectures have elaborate `AGUs`, which allow adding/subtracting an immediate value (*offset*) to/from an `AR`. For the case of embedded processors, offsets are usually stored into registers, instead of encoded in the instruction, so as to save bits for the instruction encoding. This allows instructions to be encoded into very short formats. This is the addressing mode encountered in most embedded programs.

Assume that a sequence of array references, as shown in Figure 1, occurs within the body of a loop statement. Consider also, for the sake of simplicity, that the loop body contains a single basic block and that no control statements (e.g. **if-then-else**) are allowed inside the loop. This constraint is not a requirement of the technique though, as it will be shown later (Section 4.3). Assume also that the loop induction variable i is linearly updated by an integer quantity, and that the indices of the array references within the loop are affine functions of i, e.g. $f(i) = a * i + b$, a and b integers. These assumptions are not serious restrictions, given that the majority of the array references in loops satisfy that. In the code generation approach adopted here, the piece of *Abstract Syntax* tree corresponding to an array reference is not decomposed into its atomic operations. In other words, the references to array elements are maintained until the final schedule is performed in the program. Assume also that *Common Subexpression Elimination* (CSE) is not allowed for array indices, and that *induction variable elimination* is used to optimize the loop. Induction variable elimination is an important loop optimization based on *strength reduction* and *code motion* [Aho et al. 1988]. Consider for example array reference `vector[i*a + k]`. After induction variable elimination is performed, the array element address can be computed simply by adding a to register `AR`, which is initialized to `&vector[0] + k` and hoisted outside the loop. In the case of auto-increment (decrement), i.e. `a = 1`, the `AGU` automatically increments `AR`. When multidimensional array references are present within nested loops, references can usually be reduced to the simple unidimensional case (through induction variable elimination), provided the array is laid down in memory as it is referenced in the inner loop. The goal is to rewrite array references, such that they are made dependent on a single induction variable, corresponding to its closest nested loop.

3. THE INDEXING GRAPH

The ability of a program to use the `AGU` features can be measured by the *Indexing Graph* (IG). The IG seeks to identify opportunities for array references to use indirect addressing. Once these

opportunities are found, the IG is employed to allocate an AR to each reference. Consider, for example, the loop statement in Figure 1(a). Assume that the target AGU provides auto-increment (decrement) addressing mode. The code fragment of Figure 1(a) is a typical example of an embedded application loop, which will be used in the remainder of this section. Each array reference is defined, from now on, by its *access*, defined below.

Definition 1. Let $access(r) = n$ be the function which maps a reference to an array element r into n ($n = 1, 2, \ldots$), where n is the order of r in the code sequence resulting after the instructions in the loop body have been scheduled. We say that n is the access of the array element referenced by r.

Definition 2. Let n_1 and n_2 be array accesses. Access n_1 (n_2) is said smaller (larger) than n_2 (n_1), denoted by $n_1 < n_2$ ($n_1 > n_2$), if and only if n_1 (n_2) precedes n_2 (n_1) in the the schedule order.

```
for (i = 2; i < N + 2; i++)
{
  a [ i ]   = a [i - 2] + a [i + 1] - a[i - 1];
  a [i - 1] = 2 * a [i + 2];
}
```

(a)

```
for (i = 2; i < N + 2; i++)
{
  a [i - 2]      (1)
  a [i + 1]      (2)
  a [i - 1]      (3)
  a [ i ]        (4)
  a [i + 2]      (5)
  a [i - 1]      (6)
}
```

(b)

Fig. 1. (a) Typical loop in an embedded program; (b) Array access sequence after scheduling the instructions in the loop body.

Example 1. Figure 1(b) represents the loop statement of Figure 1(a) after all instructions in the loop body have been scheduled. In order to simplify the analysis, Figure 1(b) shows only the array references and not the instructions that use them. A number (in parenthesis) is associated to each reference, corresponding to the order in which the reference occurs in the final schedule. This number is the access of that reference. For example $access(a[i+1]) = 2$.

The goal here is to allocate an AR to each array access in the loop. Observe that we seek to allocate the minimum number of ARs which can address all accesses within the loop. In this case, it is desirable to maximize the number of accesses that can share a single AR. In order to identify the possibility of sharing between two accesses, the concept of *indexing distance* is introduced.

Definition 3. Let n_1 and n_2 be array accesses and *step* the increment of the loop containing these accesses. Let $index(n)$ be a function which takes access n and returns the index associated with that access. The indexing distance between accesses n_1 and n_2 is the positive integer:

$$d(n_1, n_2) = \begin{cases} |index(n_2) - index(n_1)| & \text{if } n_1 < n_2 \\ |index(n_2) - index(n_1) + step| & \text{if } n_1 > n_2. \end{cases}$$

Example 2. Consider for example the array accesses of Figure 1(b), where $step = 1$. The indexing distance $d(1, 4) = |i - (i - 2)| = 2$. Since the indexing distance is larger than one, no auto-increment (decrement) operation can be used to update the address register allocated to access (1), such that it ends up pointing to the data requested by access (4). On the other hand, since $d(4, 1) = 1$, an

auto-decrement operation can be used to redirect the address register associated to access (4) such that it points to access (1). Notice that this will occur when access (1) is reached from access (4), across two consecutive loop iterations.

Definition 4. An *Indexing Graph* (IG) is a directed graph where each vertex corresponds to an array access, and there exists an edge (n_1, n_2) if and only if $d(n_1, n_2) \leq |k|$, where k is a post-modifier increment available in the AGU ($k = +1(-1)$, for auto-increment (decrement) mode).

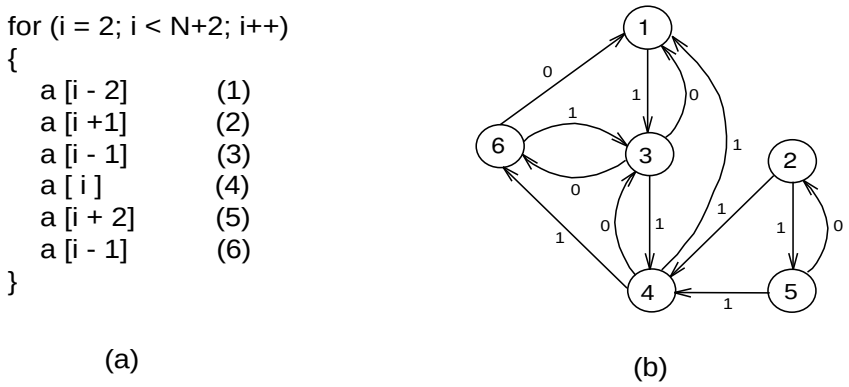

Fig. 2. (a) Typical loop in an embedded program; (b) Corresponding IG.

There exists an edge (n_1, n_2) in the IG when `AGU` operations can be used to update the address register associated to access n_1, such that it ends up pointing to access n_2. The IG has two types of edges. An edge (n_1, n_2) is a *forward edge* (or simply an *edge*) if $n_1 \leq n_2$, otherwise it is a *backward edge*.

Example 3. The IG of Figure 2(b) was built from the array accesses in the body of the loop of Figure 2(a). The IG is not a labeled graph, the labels on the edges in Figure 2(b) are used only to illustrate the indexing distance from the source to the destination vertex of each edge. Observe that edges with indexing distance one, like $(3, 4)$, capture the possibility for auto-increment (decrement) between array accesses in scheduling order. Backward edges, e.g. $(6, 1)$, identify auto-increment (decrement) operations which can be performed across loop iterations.

4. ARRAY INDEX ALLOCATION

Array Index Allocation is the problem of allocating address registers to array accesses within loops, such that the total number of address registers is minimized. In order to formalize the array index allocation problem, the concepts of IG *path* and *cycle* have to be defined.

Definition 5. A *path* $n_i \rightarrow n_j$, in the IG G is a sequence of distinct array accesses $(n_i, n_{i+1}, \ldots, n_j)$, such that (n_k, n_{k+1}) is an edge in G and $n_k < n_{k+1}$, for $i \leq k \leq j-1, i, j$ and k integers.

Definition 6. A *cycle* in the IG G is a sequence of vertices $(n_i, n_{i+1}, \ldots, n_j, n_i)$ such that $(n_i, n_{i+1}, \ldots, n_j)$ forms a path in G, for $i, j > 0$ integers.

A path in the IG corresponds to the allocation of the same `AR` to a sequence of array accesses, which can use auto-increment (decrement) addressing. Similarly, a cycle indicates that the same `AR` can be used not only for accesses in program order, but also for one more access in the next loop iteration.

Definition 7. A *cover* $C = \{c_1, c_2, \ldots\}$ of IG G is a set of disjoint paths and/or cycles in G, such that for all vertices $n \in G$, $n \in c_i$, where c_i is an element of C. The cover is disjoint, meaning that for every pair of elements c_1 and $c_2 \in C$, $c_1 \cap c_2 = \phi$.

4.1 IG Covering

The problem of minimizing the number of address registers given an IG can be formulated as the following optimization problem.

[**IG Covering**] *Given an IG G, determine the disjoint* **path/cycle** *cover of G which minimizes the total number of paths and cycles, and which has the* **smallest number of paths**. *Assume for the purpose of this problem that a vertex is a degenerated cycle of zero length.*

Each path and cycle in the cover corresponds to an address register. Moreover, unlike the `AR` associated to a path, an `AR` associated to a cycle allows auto-increment (decrement) operations to occur across consecutive loop iterations. Notice that not all cycles are allowed in the definition above. According to Definition 6 an IG cycle can only contain a single backward edge. The reason is that IG cycles should allow auto-increment (decrement) operations across a single iteration and not across multiple iterations, reflecting what occurs during the loop execution. An IG vertex is considered a cycle of zero length. Hence, a cover can contain elements which are single IG vertices. In this case, the auto-increment operation occurring at that access updates the `AR` to be used by the same access in the next iteration.

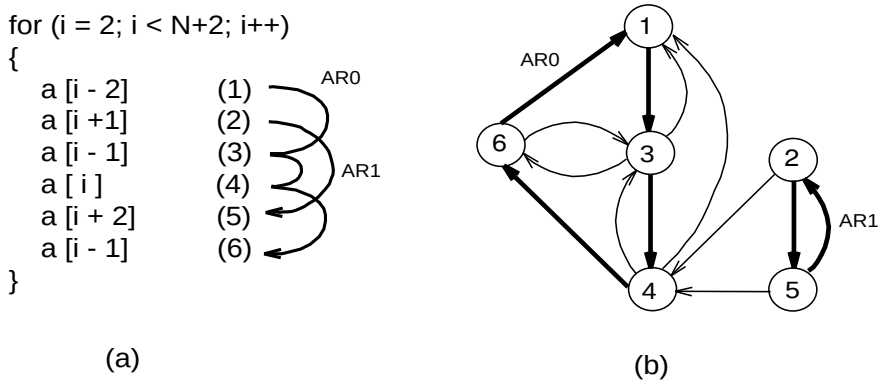

Fig. 3. (a) Typical loop; (b) Corresponding covered IG.

Example 4. Consider the IG shown in Figure 3(b). Covering the IG in that case produces a two cycle cover (and no paths) which is represented in bold. Each cycle corresponds to an address register ($\texttt{AR}_0$ and $\texttt{AR}_1$). The allocation of register $\texttt{AR}_1$ to cycle (2,5,2) takes care of the case of updating the address of the array reference (2) across loop iterations. The same is also true for register $\texttt{AR}_0$.

The solution for the IG covering problem aims to find a cover which has the smallest number of paths (or the largest number of cycles) from all the covers that have the same (minimum) cardinality. In order to understand why, let the solution of the IG Covering problem be such that a path p results. Let $head(p)$ and $tail(p)$ be the head and the tail of p. Now assume that the indexing distance from the tail to the head of p, i.e. $d(head(p), tail(p))$, is larger than one. Therefore, neither auto-increment nor auto-decrement can be used to redirect `AR` from tail to head. In this case, an instruction has to

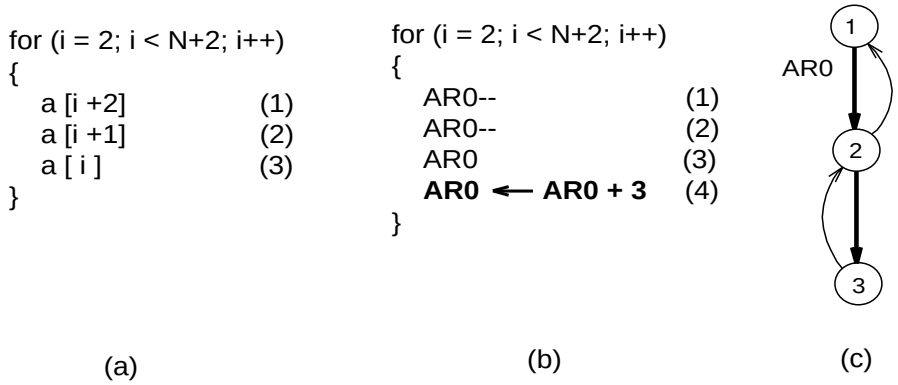

Fig. 4. (a) The original array access sequence; (b) If only auto-increment (decrement) is allowed for the access, then we need to add 3 to redirect $\texttt{AR}_0$ to point to the first access in the next iteration; (c) The corresponding covered IG.

be used in order to explicitly update the contents of the AR at $tail(p)$, such that it ends up pointing to $head(p)$.

Consider, for example, the program fragment of Figure 4(a), and its covered IG in Figure 4(c). Since the AGU provides only auto-increment (decrement) mode, and $d(3,1) = |(i+2)-(i)+1| = 3 > 1$, then no edge exists from (3) to (1). The IG cover results in a single path $(1,2,3)$, which is allocated to $\texttt{AR}_0$. At the end of the loop, $\texttt{AR}_0$ has to be redirected to point to $a[i+2]$ in the next iteration. But since no edge exists from (3) to (1), then an instruction (4) has to be added at the bottom of Figure 4(b) to explicitly do that. Therefore, the solution of the IG Covering problem should be restricted to those covers which have the smallest cardinality and the smallest number of paths, so as to minimize the number of instructions required to update the ARs at the tail of the paths.

The IG Covering problem is similar to the minimum disjoint cycle cover of a graph (MDCC). The number of disjoint cycles which cover the vertices of a graph is known as the *Hamiltonian cycle index*. Determining the minimum Hamiltonian cycle index of a graph has been shown to be NP-complete [Garey and Johnson 1979]. Cycles in a cover for the MDCC problem, unlike cycles for IG Covering, can contain more than one backward edge. Therefore, it is expected that MDCC is NP-hard, since it is as difficult as determining the minimum Hamiltonian cycle index.

4.2 The Simple IG Covering

Given that IG Covering is NP-hard we consider heuristics to tackle this problem. The most obvious one is to formulate the problem such that auto-increment (decrement) operations across loop iterations are not permitted, as in Figure 5. As a result of that, the IG becomes acyclic and the original problem is reduced to the one of determining the minimum vertex-disjoint path covering of a graph (MDPC). Based on that one can now formulate a simple version of the IG Covering problem.

[**Simple IG Covering**] *Given an IG G determine the disjoint* **path** *covering of G which* **minimizes the total number of paths**. *Assume for the purpose of this problem that a vertex is a degenerated path of zero length.*

The MDPC problem has been studied before by Boesch and Gimpel [1977]. The solution they propose is based on the Hopcroft-Karp $O(n^{5/2})$ solution of the bipartite matching problem [Hopcroft and Karp 1973]. The main idea is to split each vertex n of graph G into two vertices n_1 and n_2. Vertex n_1 (n_2) is the source (destination) of all outgoing (incoming) edges from (into) n. By doing so, the resulting graph G' becomes bipartite, with n_1 and n_2 belonging to disjoint sets. When the

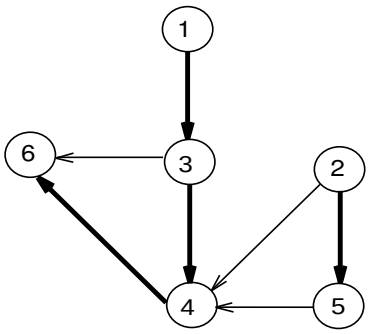

Fig. 5. Solving the MDPC for an acyclic IG.

bipartite matching algorithm is used in G' no vertex n in G' will have more than one outgoing (incoming) edge. Therefore, the matched edges in G' form a cover in G, which has only disjoint paths. Assume here that a vertex is a degenerated path of length zero. During the minimum bipartite matching all edges have the same weight. In this case, the algorithm results in the minimum number of disjoint paths which cover G.

Example 5. Solving the MDPC for the acyclic IG of Figure 5 results in paths $(1, 3, 4, 6)$ and $(2, 5)$. Cycles can still be identified in this case by computing the indexing distance between the tail and the head of a path. For example, since $d(6, 1) = 0 \leq 1$ ($d(5, 2) = 0 \leq 1$), then register $\mathtt{AR}_0$ ($\mathtt{AR}_1$) can be allocated at the tail of its corresponding path, to point to the data accessed at the head of the path.

In this particular case, the heuristic approach produces the same result as the exact solution in Section 4. This might not always be the case though. Nevertheless, the experiments in Section 5 demonstrate that Simple IG Covering produces effective improvement, and hence the need for a more expensive solution might not compensate the computational effort.

4.3 Dealing with If-then-else statements

In the previous sections it was assumed that the loop body contains a single basic block. Consider now that **if-then-else** statements are allowed within the loop, like in the program fragment of Figure 6(a). Array accesses $a[i]$, $a[i+1]$ and $a[i-1]$ are located in three different basic blocks as shown in Figure 6(b). The basic block which contains access $a[i]$ (B_1) is always executed. On the other hand, basic block B_3 (B_5), containing access $a[i+1]$ ($a[i-1]$), is executed only if variable *test* is larger (smaller/equal) than zero.

The corresponding indexing graph (without backward edges) has three vertices (1,3 and 5), one for each array access, and two edges $(1, 3)$ and $(1, 5)$, as shown in Figure 6(c). Applying IG Covering to it results in one of the two edges being covered. Assume that address register $\mathtt{AR}_1$ is allocated to the covered edge. Therefore, $\mathtt{AR}_1$ will be allocated to access (1). If the covered edge is $(1, 3)$ $((1, 5))$, then $\mathtt{AR}_1$ will also be allocated to access (3) ((5)). After $a[i]$ is accessed in B_1, $\mathtt{AR}_1$ is incremented (decremented) in order to enable $\mathtt{AR}_1$ to point to $a[i+1]$ ($a[i-1]$) during the execution of the next basic block. The IG covering algorithm has no way to distinguish which edge from these two should be covered. It is reasonable to expect that the edge associated to the most frequently executed path should have higher priority during the covering. The reason is that this increases the probability that a frequently executed path will have its addressing operations allocated to an addressing register. In order to capture that, the IG covering problem can be modified, such that IG edges associated to paths in the program are annotated with the probability of the path being taken. For example, in Figure 6(c) the edge associated to the loop condition being true (false) is labeled with probability p_T (p_F). The covering algorithm is then modified, such that the list of IG edges selected for the

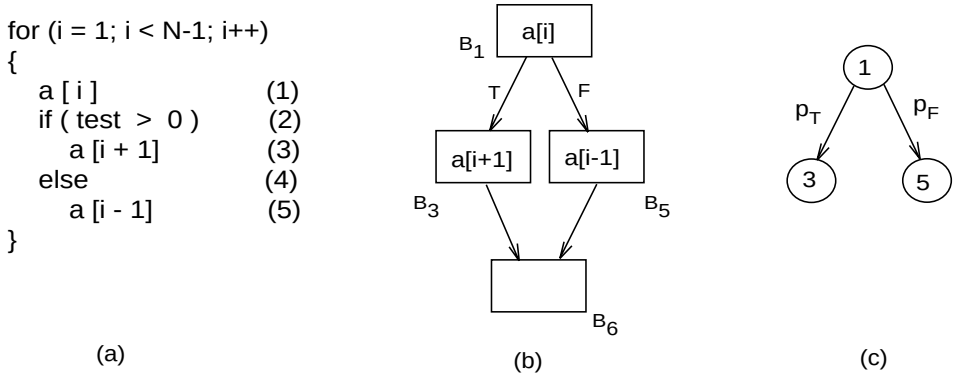

Fig. 6. (a) Loop body containing **if-then-else**; (b) The CFG representation; (c) The corresponding IG with path probabilities.

cover are sorted in a decreasing probability order. Hence, frequently exercised paths have a higher probability of being covered. On the other hand, if the array references in blocks B_3 and B_4 are the same, then both references could be allocated to the same address register, given that they refer to the same array element.

5. EXPERIMENTAL RESULTS

Array index allocation was tested using the *DSPstone* benchmark [Zivojnovic et al. 1994]. DSPstone is a well-known benchmark for DSP applications, containing typical programs found in embedded applications. The fact that it is based almost exclusively on kernels (unlike *SPEC CINT95*) reflects a characteristic of embedded systems, in which kernels play an important role. Precisely due to the lack of effective compiler optimizations for this problem, array references in DSPstone programs are performed using pointer operations. Hence, DSPstone programs had to be rewritten into their original array style. The programs were compiled using the *Texas Instruments TMS320C25 Optimizing C Compiler*. The TMS320C25 is a well-known DSP architecture. Programs were also compiled using the array index allocation technique from Section 4, which was implemented inside the SPAM compiler [SPAM 1998]. The SPAM compiler is an optimizing compiler, based on SUIF [SUIF 1998], and targeted to embedded processors. The maximum number of simultaneously live `ARs` in each program was measured, as well as the number of cycles of the final code (Table I). As expected, the largest number of simultaneously live `ARs` occurred inside inner loops which have many array references. The TI compiler uses a *Greedy* strategy to allocate address registers based on identifying sequences of references which can share a single `AR`. This strategy does not capture the possibility of sharing `AR` between non-consecutive array references. This is done by the *IG* covering algorithm in the SPAM compiler. As shown in Table I, array index allocation, based on the IG, allocates fewer `ARs` in 4 out of 10 programs. As expected it never allocates more registers than the *Greedy* algorithm.

In some programs (e.g. *biquad_N*) the TI compiler was not able to allocate all references to registers, and references were spilled into memory, resulting in a large performance penalty. The difference in performance when comparing the code generated by both compilers cannot be explained only based on the use of array index allocation. The SPAM compiler provides other optimizations which are not available in the TI compiler [SPAM 1998], but these optimizations do not affect the final number of allocated `ARs`.

Item	Program Name	Number of ARs		Cycle count	
		Greedy	IG	Greedy	IG
1	biquad_N	4	2	935	622
2	convolution	2	2	289	275
3	dot_product	2	2	107	110
4	fir	3	2	428	379
5	fir2dim	4	3	4035	2802
6	lms	2	2	763	715
7	matrix1	4	3	17897	17935
8	matrix_1x3	3	3	134	140
9	n_real_updates	4	4	711	627
10	n_complex_updates	4	4	1603	1598

Table I. Experiments with array index allocation: number of `ARs` and cycle count for *Greedy* (TI) and *IG* based allocation (SPAM).

6. CONCLUSIONS

Array index allocation, based on IG covering, minimizes the number of address registers required for array references inside loops. Preliminary results of this work have been presented in [Araujo et al. 1996]. In the meantime additional work has extended the approach proposed there. Leupers et al. [1998] described a heuristic to handle the case of covering the indexing graph with cycles and paths. Gebotys [1997] describes a technique which minimizes the cost of merging paths in the cover. Her approach is based on the solution of a minimum network flow circulation problem [Tarjan 1983]. As in the IG, every access in the loop is associated to a vertex in the graph. Every edge is associated to an unitary *edge capacity* (corresponding to a single `AR`) and a *cost*. The cost is zero (one) if no (one) instruction is needed to redirect the `AR` pointing to the source of the edge, such that it points to its destination. A *circulation edge*[1] is added to the network graph with capacity equal to the number of `ARs` available in the architecture. The idea behind the circulation edge is to limit the number of registers that can be used in a cover. The goal of the algorithm is to find the maximum circulation flow which minimizes the number of update instructions (i.e. flow cost). The drawback of this approach comes from the fact that it seeks to determine the minimum cost of the maximum flow. In other words, even if a single `AR` would be required to cover the IG with zero cost, Gebotys algorithm tries to allocate as much registers as possible, provided it does not exceed the number of registers available in the architecture. This can result in a large number of spills if the loop is nested and/or if interprocedural register allocation is performed. The reason is that registers are wasted in the loop, when they could have been minimized. An important extension of the solution proposed in this paper is certainly a combination of IG covering and Gebotys approach. The goal of a combined solution should be to minimize the number of `ARs` as much as possible, while inserting update instructions to merge paths whenever the number of allocated registers is larger than the number of registers in the architecture.

REFERENCES

Aho, A. and Johnson, S. 1976. Optimal code generation for expression trees. *Journal of the ACM 23,* 3 (July), 488–501.

Aho, A., Johnson, S., and Ullman, J. 1977a. Code generation for expressions with common subexpressions. *Journal of the ACM 24,* 1 (January), 146–160.

Aho, A., Johnson, S., and Ullman, J. 1977b. Code generation for machines with multiregister operations. In *Proc. 4th ACM Symposium on Principles of Programming Languages.* 21–28.

Aho, A., Sethi, R., and Ullman, J. 1988. *Compilers, Principles, Techniques and Tools.* Addison Wesley, Boston.

[1] A circulation edge in a network graph is an edge from the destination to the source vertex.

Appel, A. and Supowit, K. 1987. Generalizations of the Sethi-Ullman algorithm for register allocation. *Software – Practice and Experience 17,* 3 (June), 417–421.

Araujo, G., Sudarsanam, A., and S., M. 1996. Instruction set design and optimizations for address computation in DSP processors. In *9th International Symposium on Systems Synthesis.* IEEE, 31–37.

Bartley, D. H. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. *Software Practice and Experience 22,* 2 (February), 101.

Boesch, F. and Gimpel, J. 1977. Covering the points of a digraph with point-disjoint paths and its application to code optimization. *Journal of the ACM 24,* 2 (April), 192–198.

Bradlee, D., S.J., E., and Henry, R. 1991. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* 122–131.

Briggs, P., Cooper, K., Kennedy, K., and Torczon, L. 1982. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation.* 98–105.

Bruno, J. and Sethi, R. 1975. The generation of optimal code for stack machines. *Journal of the ACM 22,* 3 (July), 382–396.

Bruno, J. and Sethi, R. 1976. Code generation for one-register machine. *Journal of the ACM 23,* 3 (July), 502–510.

Callahan, D. and Koblenz, B. 1991. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation.* 192–202.

Chaitin, G. 1982. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction.* 98–105.

Chow, F. and Hennessy, J. L. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst. 12,* 4 (October), 501–536.

Fisher, C. and Kurlander, S. 1996. Minimum cost interprocedural register allocation. In *ACM Principles of Programming Languages Conference.*

Garey, M. and Johnson, D. 1979. *Computers and Intractability.* W. H. Freeman and Company, New York.

Gebotys, C. 1997. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer-Aided Design.* IEEE, 100–103.

Goodman, J. and Hsu, A. 1988. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 Conference on Supercomputing.* 442–452.

Hitchcock III, C.Y. 1986. Addressing modes for fast and optimal code generation. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

Hopcroft, J. and Karp, R. 1973. An $n^{5/2}$ algorithm for maximum matchings in bipartide graphs. *SIAM J. Comput, 2,* 4 (December), 225–230.

Horwitz, L., Karp, R., Miller, R., and Winograd, S. 1966. Index register allocation. *Journal of the ACM 13,* 1 (January), 43–61.

Lal, G. and Appel, A. 1997. Iterated register coalescing. *ACM Trans. Programming Language and Systems 18,* 3 (May), 300–324.

Leupers, R., Basu, A., and Marwedel, P. 1998. Optimized array index computation in DSP programs. In *Proceedings of the ASP-DAC.* IEEE.

Liao, S., Devadas, S., Keutzer, K., and Wang, A. 1996. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems 18*, 235–253.

Sethi, R. 1975. Complete register allocation problems. *SIAM J. Computing 4,* 3 (September), 226–248.

Sethi, R. and Ullman, J. 1970. The generation of optimal code for arithmetic expressions. *Journal of the ACM 17,* 4 (October), 715–728.

SPAM. 1998. The SPAM Project. http://ee.princeton.edu/spam.

SUIF. 1998. The SUIF Project. http://suif.stanford.edu/suif.

Tarjan, R. E. 1983. *Data Structures and Network Algorithms.* SIAM.

Zivojnovic, V., Velarde, J., and Sclåager, C. 1994. DSPstone, a DSP benchmarking methodology. Technical report, Aachen University of Technology. August.