Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures

Ashok Sudarsanam
Department of Electrical Engineering
Princeton University

Stan Liao
Advanced Technology Group
Synopsys, Inc.

Srinivas Devadas
Department of EECS
MIT

Abstract—Many application-specific architectures provide indirect addressing modes with auto-increment/decrement arithmetic. Since these architectures generally do not feature an indexed addressing mode, stack-allocated variables must be accessed by allocating address registers and performing address arithmetic. Subsuming address arithmetic into auto-increment/decrement arithmetic improves both the performance and size of the generated code.

Our objective in this paper is to provide a method for comprehensively analyzing the performance benefits and hardware cost due to an auto-increment/decrement feature that varies from $-l$ to $+l$, and allowing access to k address registers in an address generator. We provide this method via a parameterizable optimization algorithm that operates on a procedure-wise basis. Hence, the optimization techniques in a compiler can be used not only to generate efficient or compact code, but also to help the designer of a custom DSP architecture make decisions on address arithmetic features.

We present two sets of experimental results based on selected benchmark programs: (1) the values of l and k beyond which there is little or no improvement in performance, and (2) the values of l and k which result in minimum code area.

I. Introduction

Microprocessors such as microcontrollers and fixed-point digital signal processors (DSPs) are increasingly being embedded into many electronic products. Two main trends in the design of embedded systems are becoming clear: (1) the amount of embedded software is growing increasingly larger and more complex, and (2) all the electronics—microprocessor, RAM, ROM, and ASICs—are being incorporated into a single integrated circuit.

Since program code resides in on-chip ROM, the size of this code translates directly into silicon area and cost. It is therefore an important problem to minimize the size of the program code, while simultaneously optimizing performance. However, current compilers for microcontrollers and fixed-point DSPs generate code that is extremely unsatisfactory with respect to code size and performance [8]—thus, programming in a high-level language can incur penalties on code size and performance. We believe that generating the best code for embedded processors will require not only traditional optimization techniques (e.g., [1]), but also new techniques that take advantage of special architectural features that decrease code size. Our recent efforts in this direction are summarized in [7].

Many architectures, such as the VAX, Texas Instruments TMS320C25, and most embedded controllers, provide indirect addressing modes with auto-increment/decrement[1] arithmetic. These features allow for efficient sequential access of memory and increase code density, since they subsume address arithmetic instructions and result in shorter instructions in variable-length instruction architectures. In particular, DSPs and embedded controllers are designed assuming that software that runs on them would make heavy use of auto-increment addressing. In many cases, DSPs and controllers do not provide an addressing mode for indexing with a constant offset. Thus, automatic variables (those variables that are dynamically allocated on the stack frame) must be accessed by allocating address registers and performing address arithmetic. Subsuming address arithmetic into auto-increment arithmetic improves both the size and performance of the generated code.

Determining the storage assignment of automatic variables so as to maximize the use of auto-increment arithmetic has received much recent attention. Bartley [2] was the first to address a simple version of the *offset assignment problem*, and presented an approach based on finding a maximum-weight Hamiltonian path of an access graph. Offset assignment problems for unit increments on multiple address registers have been addressed by Liao et al. [6] and Leupers [5]. While effective algorithms have been proposed, the machine model in these works is quite restrictive and does not reflect the address arithmetic capabilities of some of the more advanced DSP architectures. For instance, the parallel processor in the *Texas Instruments TMS320C80* features eight address registers and allows for increments ranging from -7 to $+7$. In this paper, we extend previous work by permitting multiple address registers with arbitrarily-large increments.

The motivation for this paper is based on the following scenario: suppose that a designer of a custom DSP architecture, given a set of applications, wishes to determine the number of address registers and the range of auto-increment for the address generation unit. Increasing the number of address registers k may improve the execution speed and/or static instruction count. Furthermore, enlarging the range of increment $[-l, l]$ may also reduce the number of explicit address arithmetic instructions. However, increasing either k or l may increase the number of bits required to encode the instruction word. Our objective in this paper is to provide a method for comprehensively analyzing the performance benefits and hardware cost (in terms of code *area*) due to an auto-increment feature that varies from $-l$ to $+l$, and allowing access to k address registers in an address generator (we use the term *code area* to emphasize the fact that the overall code size depends not only on the number of instructions, but also the width of the instruction

[1]Henceforth, *increment* will be used to represent both increment and decrement.

DAC 97, Anaheim, California

word). We provide this method via a parameterizable optimization algorithm that operates on a procedure-wise basis. Thus, we illustrate the important role an optimizing compiler plays in architecture design. To this end, we present a generalized formulation of the storage assignment problem, parameterized by the maximum allowable increment l and the number of address registers k.

This paper is organized as follows: in Sec. II, we give an example illustrating the *simple offset assignment* problem with increment range $[-l, l]$ (which we call l-SOA); in Sec. III, we present a generalization of the SOA algorithm given in [6] for l-SOA; in Sec. IV, we briefly generalize the *general offset assignment* problem with increment range $[-l, l]$ (which we call (l, k)-GOA); in Sec. V, we describe how an offset assignment for an entire procedure may be determined; Sec. VI provides experimental results—two highlights of these results are (1) the values of l and k beyond which there is little or no improvement in performance, and (2) the values of l and k which result in minimum total code area; we finally present our conclusions in Sec. VII.

II. Example

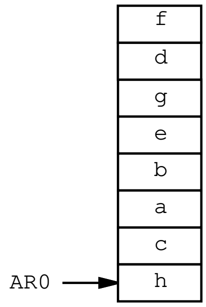
We assume that the reader is familiar with the simple offset assignment (SOA) problem presented in [6]. In this section, we present an extension of the SOA problem that takes into account the capability of increments ranging from $-l$ to $+l$, for arbitrary l. Although our optimizations are applied on a procedure-wise basis, we focus on *basic blocks* in this section.

Fig. 1(a) shows an example code sequence and Fig. 1(b) shows the optimal offset assignment for the variables in this sequence, assuming that a single address register with unit auto-increment is used. Address register `AR0` is used to address the variables; `ADAR` (`SBAR`) adds a constant to (subtracts a constant from) `AR0`; `LDAR` loads `AR0` with an address; finally, `*(AR0)+` and `*(AR0)-` denote auto-increment and auto-decrement by one, respectively, of `AR0`. The instructions we intend to minimize are precisely the explicit address arithmetic instructions, namely `ADAR` and `SBAR`. As highlighted in Fig. 1(c), six such instructions are required, given the offset assignment of Fig. 1(b).

Now if $l = 2$, then variables that are separated by a distance of at most two may be accessed in sequence without requiring an explicit address arithmetic instruction. Fig. 2(b) and Fig. 2(c) show the optimal offset assignment and assembly code, respectively, corresponding to the code sequence of Fig. 2(a). Here, the notation `*(AR0)`$+j$ denotes auto-increment of `AR0` by j. Note that the additional capability afforded by increasing l reduces the number of `ADAR` and `SBAR` instructions, since adjacent accesses to variables that are distance-l apart in the offset assignment incur no penalty.

```
e = a + b;
b = b + g;
c = c + d + f;
a = h - c;
c = b - c + a;
```

(a)



(b)

```
LDAR   AR0,&a     ; a
LOAD   *(AR0)+    ; b
ADD    *(AR0)+    ; e
STOR   *(AR0)-    ; b
LOAD   *(AR0)
ADAR   AR0,2      ; g
ADD    *(AR0)
SBAR   AR0,2      ; b
STOR   *(AR0)
SBAR   AR0,2      ; c
LOAD   *(AR0)
ADAR   AR0,5      ; d
ADD    *(AR0)+    ; f
ADD    *(AR0)
SBAR   AR0,6      ; c
STOR   *(AR0)-    ; h
LOAD   *(AR0)+    ; c
SUB    *(AR0)+    ; a
STOR   *(AR0)+    ; b
LOAD   *(AR0)
SBAR   AR0,2      ; c
SUB    *(AR0)+    ; a
ADD    *(AR0)+    ; c
STOR   *(AR0)
```

(c)

Fig. 1. (a) Code sequence (b) Optimal offset assignment for $l = 1$ (c) Assembly code

III. Simple Offset Assignment Problem

In this section, we assume that a single address register is available to address the automatic variables of each procedure. We term this the *l-simple offset assignment* (l-SOA) problem. We first review the formulation of [6] for the $l = 1$ case, and then extend it to the case where $l > 1$. We will subsequently refer to this formulation as the 1-SOA problem, and the more general formulation as the l-SOA problem.

A. Access Sequence and Access Graph

Given a code sequence for a basic block, we can uniquely define an *access sequence* for this block. For an operation $z = x$ **op** y, the access sequence is $x\,y\,z$. The access sequence for an ordered set of operations is simply the concatenated access sequences for each operation in the appropriate order. The access sequence for the basic block of Fig. 1(a) is shown in Fig. 3(a).

Given the notion of an access sequence, it can be seen that when $l = 1$, the cost of an assignment is equal to the number of adjacent accesses of variables that are not assigned to adjacent stack locations. For instance, six address arithmetic instructions are required for the offset assignment of Fig. 1(b), since the following six two-symbol substrings of the access sequence refer to variables assigned to non-adjacent stack locations: `b g`, `g b`, `b c`, `c d`, `f c`, and `b c`.

The *access graph* $G\langle V, E\rangle$ is derived from an access sequence as follows: each node $v \in V$ corresponds to a unique variable, and edge $e\langle u, v\rangle \in E$ exists with weight $w(e)$ if variables u and v are adjacent to each other $w(e)$ times in the access sequence. The access graph for the basic block of Fig. 1(a) is shown in Fig. 3(b).

With respect to the access graph, if $l = 1$, then the cost of an assignment is equal to the sum of the weights of all edges that connect variables assigned to non-adjacent stack locations. For the example of Fig. 1, $\langle \mathtt{b}, \mathtt{g}\rangle$, $\langle \mathtt{b}, \mathtt{c}\rangle$, $\langle \mathtt{c}, \mathtt{f}\rangle$, and $\langle \mathtt{c}, \mathtt{d}\rangle$ are such edges, and these have a total weight of 6. The other edges connect variables that have been assigned to adjacent stack locations and thus, incur no cost.

B. 1-SOA and the MWPC Problem

Definition 1: A *disjoint path cover* (henceforth *cover*) of a weighted graph $G\langle V, E\rangle$ is a subgraph $C\langle V, E'\rangle$ of G such that:

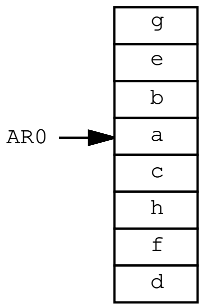
- for each $v \in V$, $\deg(v) \le 2$ (degree constraint);
- there are no cycles in C (cycle constraint).

Note that the edges in C form a set of disjoint paths (some of which may contain no edges), hence the name.

Definition 2: The *weight* of a graph G is the sum of the weights of all edges in G. The *cost* of a cover C of G is the sum of the weights of all edges in G, but not in C, i.e., cost(C)

```
e = a + b;
b = b + g;
c = c + d + f;
a = h - c;
c = b - c + a;
```

(a)



(b)

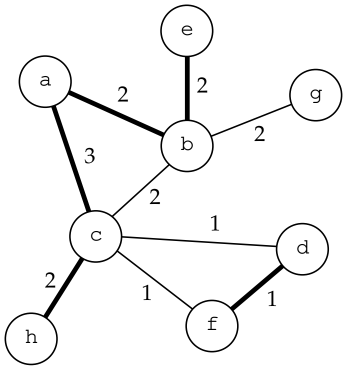
```
LDAR  AR0,&a     ; a
LOAD  *(AR0)+    ; b
ADD   *(AR0)+    ; e
STOR  *(AR0)-    ; b
LOAD  *(AR0)+2   ; g
ADD   *(AR0)-2   ; b
STOR  *(AR0)-2   ; c
LOAD  *(AR0)
SBAR  AR0,3      ; d
ADD   *(AR0)+    ; f
ADD   *(AR0)+2   ; c
STOR  *(AR0)-    ; h
LOAD  *(AR0)+    ; c
SUB   *(AR0)+    ; a
STOR  *(AR0)+    ; b
LOAD  *(AR0)-2   ; c
SUB   *(AR0)+    ; a
ADD   *(AR0)-    ; c
STOR  *(AR0)
```

(c)

Fig. 2. (a) Code sequence (b) Optimal offset assignment for $l = 2$ (c) Assembly code

a b e b g b c d f c h c a b c a c

(a)



(b)

Fig. 3. (a) Access sequence (b) Access graph

= weight(G) - weight(C).

Definition 3 (MWPC) Given an access graph G, find a cover C with maximum weight.

Note that a cover with maximum weight is equivalent to a cover with minimum cost. It has been shown in [6] that solving the MWPC problem is equivalent to solving the 1-SOA problem. Since MWPC is NP-complete, a heuristic algorithm for 1-SOA is proposed.

C. *l-SOA*

The main inadequacy of the 1-SOA formulation is that when $l > 1$, some edges of the access graph will have an effective *zero cost*, *even if they are not included in the cover*. In Fig. 3(b), if $l = 2$, then $\langle \texttt{c}, \texttt{f} \rangle$ (which contributed a cost of one in the $l = 1$ case) contributes zero cost since c and f are distance-2 apart

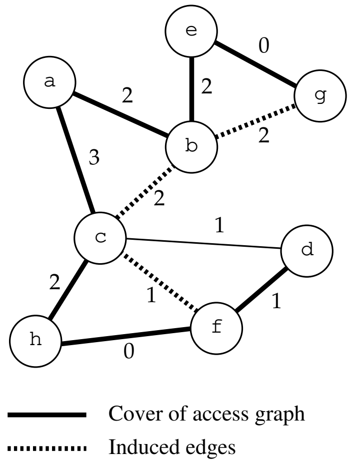


Fig. 4. A cover and its induced edges, for $l = 2$.

with respect to the final offset assignment (shown in Fig. 2(b)). When loading f, AR0 can be auto-incremented by 2 so that it points to the location for c.

Thus, we need to redefine the weight of a cover to take into consideration those edges whose costs are subsumed by virtue of l. Note that the edge-selection criteria for a disjoint path cover do not change, because our goal remains to find a linear ordering of variables; however, a different cost function must now be minimized. To accomplish this goal, we must now consider *complete graphs*, rather than graphs consisting of only positive-weight edges. The reason for this is as follows: when $l = 1$, including zero-weight edges in the cover affords no benefit, and including such edges in the access graph only makes the optimization process less efficient; however, when $l > 1$, selecting a zero-weight edge may *induce* another positive-weight edge to become effectively cost-less, even though the latter could not itself be selected due to the cycle and degree constraints. To precisely describe which edges have effectively zero cost, we introduce the notion of *induced* $(l + 1)$*-cliques*:

Definition 4: Let $G\langle V, E\rangle$ be a complete graph, and C be a cover of G. A subgraph $Q\langle V_Q, E_Q\rangle$ of G consisting of $(l + 1)$ nodes and edges connecting these nodes is said to be an *induced* $(l+1)$*-clique* of C if V_Q are nodes of some subpath in C of length l.

Definition 5: An edge of G is said to be *induced* by a cover C if it is not in C but belongs to some induced $(l + 1)$-clique of C.

Intuitively, if nodes u and v belong to an induced $(l + 1)$-clique, then their distance from each other, with respect to edges of C, is at most l. Since the address generation unit can auto-increment by at most l, the edge $\langle u, v\rangle$ incurs no cost. We define the l-induced weight of a cover as follows:

Definition 6: The l-induced weight of a cover C of a complete, weighted graph G is the total weight of the edges in C and those edges induced by C. The *l-induced cost* of C is the sum of the weights of edges that are neither part of C nor induced by C.

Definition 7 (l-SOA) The l-SOA problem consists of finding a cover of an access graph G with minimum l-induced cost.

Fig. 4 shows an example of a cover and its induced $(l + 1)$-cliques, for $l = 2$. The induced $(l + 1)$-cliques are: beg, abe, abc, ach, cfh, and dfh, and the positive-weight induced

```
SOLVE-l-SOA(L)
{
  /* L = access sequence for basic block or trace */
  G⟨V,E⟩ ← ACCESS-GRAPH(L)
  F ← list of edges in E
  C⟨V′,E′⟩ : V′ ← V, E′ ← { }
  while ( |E′| < |V| − 1 and F not empty ) {
    foreach edge e in F {
      Compute the total weight, z(e), of those
          edges that would become induced if
          e were included in C
    }
    Select edge e* which maximizes z(e) + w(e)
    E′ ← E′ ∪ e*
    Remove from F those edges made ineligible
        due to cycle and degree constraints
  }
  /* Construct an assignment from E′ */
  return CONSTRUCT-ASSIGNMENT(E′)
}
```

Fig. 5. Heuristic algorithm for l-SOA.

edges are ⟨b, g⟩, ⟨b, c⟩, and ⟨c, f⟩ (for clarity, edges with zero weight are not shown unless they are part of the cover). In this example, ⟨c, d⟩ is the only edge for which we have to pay the cost of explicit address arithmetic instructions.

Doubtless, the l-SOA problem is NP-hard. As with 1-SOA, we propose a heuristic procedure, shown in Fig. 5 to solve l-SOA: in each iteration of the edge-selection process, we compute the weight each eligible edge e *would* contribute by summing up the weights of all edges that would become induced if e were included in the current partial solution. The edge e^* with the largest contribution is selected.

IV. General Offset Assignment Problem

We briefly generalize the l-offset assignment problem to the case where k address registers, `AR0` through `AR`$(k-1)$, are available. Since this formulation involves two independent parameters, we name it the (l,k)-*general offset assignment* ((l,k)-GOA) problem. In this formulation, we assume that (1) there is a fixed setup cost of introducing the use of an address register and (2) each address register is used to point to a disjoint subset of variables.

Definition 8: Let L be the access sequence of a basic block, and V be the set of variables in L. The *access subsequence* generated by $W \subseteq V$ is the subsequence of L consisting of those variables only in W.

Definition 9 ((l,k)-GOA) Given an access sequence L, the set of variables V, and the number of address registers k, find a partition of V, $\Pi = \{P_1, P_2, ..., P_m\}$, $m \leq k$, in which the total cost of the optimal l-SOA of the corresponding access subsequences *plus* the setup costs for using m registers is minimum.

Since determining the optimal variable partition is NP-complete, various partitioning heuristics are proposed in [6]. One heuristic partitions the set of variables into subsets of cardinality two, since the cost of performing l-SOA on an access subsequence consisting of two unique variables is *zero*.

V. l-SOA for Procedures

In this section, we describe how our address assignment methods may be applied to entire procedures. We first discuss some pre-processing steps that not only reduce the data-memory requirements of the application program, but may also improve the quality of the final offset assignment. We then describe the process of constructing an access graph for a procedure.

A. Minimizing Storage Requirements

We first perform a *coloring* of each procedure's automatic variables, such that identically-colored variables are allocated to the same stack location. This optimization significantly reduces the run-time data-memory requirements of the program, which is of extreme importance in embedded systems design. It is also possible that reducing the size of the stack frame may result in an offset assignment of lower cost.

The coloring process proceeds as follows: first, *liveness analysis* [1] is performed on all variables so as to determine their *live ranges*—the live range of a variable `v` specifies those regions of the program where `v` is active. Secondly, an *interference graph* is constructed in which a vertex exists for each variable, and edge $e\langle i,j\rangle$ exists if and only if the live ranges of variables i and j overlap. This edge specifies that i and j must be colored differently, or analogously, they must be allocated to different stack locations. Finally, we apply Briggs' heuristic [3] to this graph in order to find a coloring that satisfies all interference constraints.

In order to reduce the number of edges in the interference graph and hence, possibly decrease the number of required colors, we perform a *renaming* of all variable uses and definitions prior to constructing this graph. This technique is analogous to the *register renaming* hardware scheme employed in high-performance processors [4]. Variable renaming reduces the life-times of variables and hence, possibly reduces the number of overlapping live ranges.

B. Constructing Procedural Access Graph

We can construct a procedural access graph on which the l-SOA algorithm operates, thus allowing us to obtain an offset assignment for the entire procedure. This is achieved by merging the access graphs of each basic block in the procedure, and adding weights to the edges of the resulting access graph so as to account for variable accesses along control-flow edges. However, depending on the optimization objective, it may be necessary to weight the access graph of each basic block differently: for optimization of *static* instruction count, we simply weight each basic block equally; for optimization of *dynamic* instruction count, we weight each basic block according to its expected execution frequency.

The expected execution frequencies of basic blocks can be determined statically or through the use of profiling data. Although profile-driven estimations are generally more accurate, the quality of static estimations is improving. We have implemented the static estimation method of [9] to estimate basic block frequencies. We briefly outline the steps involved in this algorithm:

- first, the type of each control-flow edge (i.e., *forward* or *backward*) is determined through the use of *dominator analysis* [1], which is computed iteratively; we then classify the type of each edge $e\langle i,j\rangle$ as follows: if node j dominates node i, then $e\langle i,j\rangle$ is a *backward* edge, otherwise it is a *forward* edge;
- using various heuristics outlined in [9], a probability is assigned to each edge based on its type;
- a system of linear equations is set up, in which the variables

correspond to basic blocks, and the coefficients are the determined edge frequencies;

- a solution to these equations is found using a linear programming solver – this solution specifies the execution frequency of each basic block.

VI. Experiments and Results

We present two sets of experimental results that demonstrate the variation in dynamic instruction count and code area across different values of l and k. All measurements were obtained by running our parameterizable offset assignment algorithm using various heuristics to partition the variables, and selecting the best results. For the first set of experiments (dynamic instruction count), the basic blocks were weighted according to their estimated frequencies, whereas for the second set, they were weighted equally.

It is important to note that basic block weighting and the two experiments are *orthogonal*. In other words, given a particular weighting, both experiments should be performed and the data collected from these experiments serve different purposes. The objectives are summarized in Table I. For instance, if our primary goal is to optimize for performance, we weight the basic blocks according to frequency, and we use the data set from the second experiment to evaluate the impact on total area. Thus, the experimental results given in this section only represent cases (1) and (4) in Table I.

We performed our experiments on core routines from the following programs: the image-processing program `xv`, an implementation of the JPEG encoder/decoder, the compression program `gzip`, and the DES and RSA cryptosystems. We grouped these routines into three sets based on the total number of automatic variables present (including front-end-generated temporaries): there were 12 small examples with fewer than 20 variables, 23 medium examples with 20–60 variables, and 4 large examples with more than 60 variables.

A. Address Arithmetic Instruction Count

We present in Fig. 6(a), Fig. 6(b), and Fig. 6(c) results indicating address arithmetic (i.e., `ADAR`, `SBAR`, and `LDAR`) instruction counts for the small, medium and large examples, respectively. In each plot, we give the minimum values of l and k necessary to reduce address arithmetic instruction count to less than 2%, 5%, and 10% of the $l = 0, k = 1$ case—in this case, an `ADAR` or `SBAR` is required for each memory access. For instance, for the small examples, to reduce the number of address arithmetic instructions to 5% or less, the pair (l, k) has to equal or exceed one of the following points: (1,3), (2,2), or (4,1). Similarly, for the large examples, to reduce the address arithmetic count to 2% or less, (l, k) has to be greater than or equal to either (4,5) or (5,3).

It is interesting to note that the largest reduction in the number of address arithmetic instructions due to an increase in k occurs at the transition from $k = 1$ to $k = 2$. Beyond that, we observe diminishing returns, largely due to the setup costs required for using additional address registers. For instance, in Fig. 6(b) for the $l = 3$ case, it is not possible to obtain much further improvement by increasing k beyond 3. However, if $k = 2$, we can reach 2% by increasing l to 5.

B. Overall Code Area

Reducing the number of address arithmetic instructions by increasing k and l incurs a cost: more bits are required in the instruction word to encode which address register to use and the increment amount. We present in Table II(a), Table II(b), and Table II(c) results indicating total code area for the small, medium and large examples, respectively. The calculations were performed as follows: we assumed a 10-bit base instruction word, and for any (l, k) pair, we computed the bit-width of the instruction word as $10 + \lceil \log_2(2l + 1) \rceil + \lceil \log_2 k \rceil$. For each (l, k) pair in each set of examples, the total number of instructions was computed by adding the number of non-`ADAR`/`SBAR` instructions to the number of address arithmetic instructions. This number is multiplied by the instruction bit-width to obtain the code area.

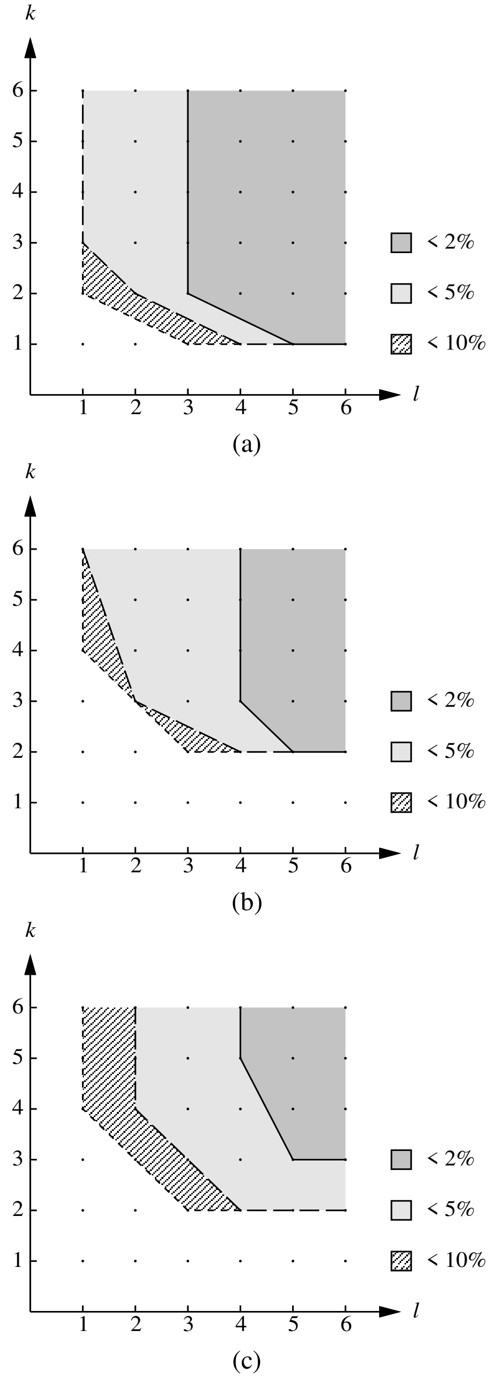


Fig. 6. Dynamic address arithmetic instruction count for (a) small, (b) medium-sized, and (c) large examples.

For the small examples, the minimum code area values occur

	Dynamic instruction count	Total code area
Frequency-weighted	Optimize for performance (1)	Evaluate impact on area (2)
Equally-weighted	Evaluate impact on performance (3)	Optimize for area (4)

TABLE I

ONE EXPERIMENT IS USED TO OPTIMIZE FOR AN OBJECTIVE (PERFORMANCE OR AREA), WHILE THE OTHER IS USED TO EVALUATE IMPACT ON THE OTHER OBJECTIVE (AREA OR PERFORMANCE). FOR INSTANCE, IF OUR PRIMARY OBJECTIVE IS TOTAL CODE AREA (4), WE WOULD CONSTRUCT THE PROCEDURAL ACCESS GRAPH BY WEIGHTING THE BASIC BLOCKS EQUALLY, AND THE MEASUREMENTS OBTAINED IN (3) WOULD THEN BE USED AS A GUIDE FOR TRADE-OFFS IN PERFORMANCE.

l	**k**					
	1	*2*	*3*	*4*	*5*	*6*
1	2.971	**2.573**	2.670	**2.647**	2.835	2.835
2	2.786	2.659	2.826	2.826	3.014	3.014
3	**2.592**	**2.633**	2.822	2.822	3.010	3.010
4	3.054	2.822	3.010	3.010	3.198	3.198
5	2.670	2.822	3.010	3.010	3.198	3.198
6	2.650	2.822	3.010	3.010	3.198	3.198

(a) $\times 10^4$

l	**k**					
	1	*2*	*3*	*4*	*5*	*6*
1	1.235	1.103	1.187	**1.030**	1.072	1.055
2	1.229	1.070	1.071	1.049	1.114	1.114
3	1.164	**1.012**	1.047	**1.044**	1.113	1.113
4	1.186	1.058	1.114	1.113	1.182	1.182
5	1.144	1.047	1.113	1.113	1.182	1.182
6	1.106	1.045	1.112	1.112	1.182	1.182

(b) $\times 10^5$

l	**k**					
	1	*2*	*3*	*4*	*5*	*6*
1	3.629	3.275	3.214	**3.080**	3.234	3.174
2	3.679	3.217	3.227	3.131	3.318	3.285
3	3.544	**3.055**	3.132	**3.078**	3.266	3.264
4	3.692	3.180	3.293	3.266	3.466	3.466
5	3.588	3.126	3.270	3.256	3.460	3.460
6	3.499	3.117	3.264	3.253	3.456	3.456

(c) $\times 10^4$

TABLE II

OVERALL CODE AREA (IN BITS) FOR (A) SMALL, (B) MEDIUM-SIZED, AND (C) LARGE EXAMPLES. THE HIGHLIGHTED NUMBERS SHOW THE BEST RESULTS.

at the following (l, k) points: (1,2), (3,1), (3,2), and (1,4)—these areas are within 2% of each other. For both the medium and large examples, the minimum code area values occur at the following (l, k) points: (3,2), (1,4), and (3,4)—these areas are within 1% of each other. Therefore, we conclude that $l = 3, k = 2$ and $l = 1, k = 4$ are the best options for a wide range of examples. Our assumption about the base instruction word is realistic for DSPs—varying the base width between 8 and 16 bits has minimal impact (qualitatively) on the results.

VII. CONCLUSIONS

We have presented a methodology for the analysis and evaluation of the impact on performance and code area due to various address generation unit capabilities. Specifically, we have examined two parameters: the number of address registers and the range of auto-increment. While increasing either of these parameters may result in potential code size or performance gains, doing so requires increasing the instruction word width. We have identified, through the use of a parameterizable optimization algorithm on a wide range of examples, the optimal values for these two parameters with respect to performance and code area. Our experimental results are particularly valuable to designers of custom DSP architectures, who must decide which features to incorporate into the address generation unit. Our methodology may also be applied to a wider or more narrow set of benchmarks, so as to obtain the best parameters for the applications for which the architecture is intended.

ACKNOWLEDGMENTS

This research was supported in part by the Advanced Research Projects Agency under contract DABT63-94-C-0053 and in part by NSF contract MIP-9612632.

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] D. H. Bartley. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software—Practice and Experience*, 22(2), February 1992.

[3] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[4] R.M. Keller. Look-Ahead Processors. *Computing Surveys*, 7(4), December 1975.

[5] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Proceedings of International Conference on Computer-Aided Design*, 1996.

[6] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage Assignment to Decrease Code Size. In *ACM Transactions on Programming Languages and Systems*, volume 18, May 1996.

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, G. Araujo, A. Sudarsanam, S. Malik, V. Živojnović, and H. Meyr. Code Generation and Optimization Techniques for Embedded Digital Signal Processors. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*. Kluwer Academic Publishers, 1996. Proc. of the NATO Advanced Study Institute on Hardware/Software Co-Design.

[8] V. Živojnović, J. Martínez Velarde, and C. Schläger. DSPstone: A DSP-oriented Benchmarking Methodology. In *Proc. of the 5th Int'l Conf. on Signal Processing Applications and Technology*, October 1994.

[9] T.A. Wagner, V. Maverick, S.L. Graham, and M.A. Harrison. Accurate Static Estimators for Program Optimization. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.