Conflict Graph Based Allocation of Static Objects to Memory Banks

Peter Keyngnaert Bart Demoen

Bjorn De Sutter Bruno De Bus Koen De Bosschere

Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Leuven, Belgium
{peter.keyngnaert,bart.demoen}@cs.kuleuven.ac.be
Department of Electronics and Information Systems
Universiteit Gent
B-9000 Gent, Belgium
{bjorn.desutter,bruno.debus,koen.debosschere}@elis.rug.ac.be

September 6, 2001

ABSTRACT

Several architectures, in particular those specifically designed for digital signal processing, have a memory structure that consists of a number of banks with different characteristics such as waitstate, size, ... There may also exist constraints on the accessibility of these banks, as some bank combinations can be accessed in parallel, while others can not. As memory access conflicts lead to pipeline stalls, the assignment of the data objects of a program to the set of memory banks is crucial with respect to a program's execution speed. Programmers usually do the assignment of the static objects manually. We present a method to automate this process at/post link-time, as the linking process is the first moment at which both the entire program as well as the target architecture's characteristics are fully known. Based upon statistics drawn from an execution trace of the program, an ordering of conflicts is derived according to the possible execution time penalties they generate. By allocating the objects of those conflicts that have the most negative impact on the program execution time first, a decent allocation can be derived automatically.

Keywords: memory allocation, static objects, DSP, memory architecture, conflict graph

1 INTRODUCTION

Some processors, especially those dedicated to digital signal processing (DSPs), have a complex memory architecture: their main memory space physically consists of a number of memory banks that may not only have different sizes but also different access constraints and access times: some (combinations of) banks may be accessed in parallel while others may not; some banks allow more than one access per cpu cycle or have a smaller waitstate than others. For this kind of processing units, data placement has severe implications on the execution speed of an application. Objects that are often accessed together during the same cpu cycle greatly benefit from being assigned to memory banks in such a way that the number of conflicts leading to pipeline stalls or the insertion of pipeline bubbles is minimal.

In the DSP world, it is still common practice to optimize the final version of a program by hand, as speed is crucial for real-time applications and humans still outperform compilers at writing the fastest code. One of the tasks not yet handled by the development environment is the aforementioned assignment of static objects to memory banks. This task is partially supported by the system, as there exist tools that let programmers drag static variables from their source code into a visual representation of the memory architecture they target (e.g. [15]), but it takes a lot of time and expertise to come up with a good placement. In this paper, we propose a novel way of automating this process.

In section 2 we formalize the problem and prove it to be *NP-complete*. Section 3 introduces the conflict graph and shows how it can be used to impose an order upon the objects by which they should be allocated to a memory bank. Section 4 then proposes some heuristics to automate the allocation process. Sections 5, 6 and 7 cover related work, considerations for future work and conclusions respectively.

2 THE PROBLEM AND ITS COMPLEXITY

Given are the set of static objects O used by a program P, as well as the set of memory banks B of the target architecture. The goal is to find an allocation function $alloc : O \to B$ that minimizes the execution time of P. In the next paragraphs we show that this problem is *NP-complete*, so our goal is to find a good approximation of the optimal solution in an acceptable amount of time.

Graph colouring Consider the well-known problem of graph colouring. Given are a graph $G = (V, E)$ and a set C where V is the set of vertices (nodes), $E \subseteq V \times V$ is the set of edges between nodes of V and C is a set of k colours. A k-colouring of G is a mapping $col : V \to C$ that colours each node in V with a colour from C where all k colours are used and taking the following restriction into account:

$$\forall n_i, n_j \in V : \exists (n_i, n_j) \in E \implies col(n_i) \neq col(n_j)$$

The problems of deciding whether such a mapping *col* exists as well as finding one are *NP-complete* [10].

A restricted version of object allocation Now consider the problem of allocating the objects used by a program P to the memory of an architecture A that supports multiple memory banks. Let O be the set of n objects used by P, let $size(o)$ be the size of object $o \in O$ and let B be the set of k memory banks of A where $size(b)$, $ws(b)$ and $acc(b)$ are the size, waitstate and possible number of accesses (per cpu cycle) to bank $b \in B$ respectively. For each memory bank $b \in B$ we assume the following characteristics:

- $size(b) = \sum_{j=1}^{n} size(o_j),\ \forall j\ (1 \leq j \leq n) : o_j \in O$
- $ws : B \to \{0\} : b \mapsto 0$
- $acc : B \to \{1\} : b \mapsto 1$

A allows any combination of 2 memory banks to be accessed concurrently during one cpu cycle. During the execution of P, at most 2 objects need to be accessed during the same cpu cycle. We call $c(o_i, o_j)$ a *conflict* between objects $o_i, o_j \in O$ if o_i and o_j need to be accessed during the same cpu cycle: there will be a loss of execution speed if they are assigned to the same memory bank. Solving the allocation problem to avoid as much conflicts as possible is equivalent to finding a k-colouring

$$col' : V' \to C'$$

of $G' = (V', E')$ where $V' = O$, $E' \subseteq O \times O$ and $C' = B$ is the set of colours. The colouring restriction

$$\forall o_i, o_j \in O : \exists (o_i, o_j) \in E' \implies col'(o_i) \neq col'(o_j)$$

holds. This problem naturally maps onto the graph colouring problem given in the previous paragraph, so this allocation problem is $NP - complete$ also.

A general version of object allocation Apply the following generalizations to the allocation problem given in the previous paragraph:

- allow only certain combinations of i $(1 \leq i \leq k)$ memory banks of B to be accessed concurrently during one cpu cycle
- allow conflicts $c(o_i, \ldots, o_{i+j-1})$ of arbitrary size j, or equivalent: allow an edge $e \in E'$ to connect any j $(2 \leq j \leq n)$ nodes $o_i, \ldots, o_{i+j-1} \in O$. j is called the *degree* of e: $j = degree(e)$
- allow memory banks to have any waitstate:
$$ws : B \to \mathbb{N} : b \mapsto ws(b)$$
- allow memory banks to have any size:
$$size : B \to \mathbb{N}_0 : b \mapsto size(b)$$
- allow memory banks to support an arbitrary number of accesses during one cpu cycle
$$acc : B \to \mathbb{N}_0 : b \mapsto acc(b)$$

The problem given in the previous paragraph is just one specific instance of the general class of problems given here. Since that one instance is *NP-complete*, solving the problem in general must also be *NP-complete*.

Graph-Program bijections The conclusion of the previous paragraph also requires that the following holds:

1. for each program P, a conflict graph graph $G = (V, E)$ exists
2. for each graph $G = (V, E)$ with $d = max\ \{degree(e) \mid e \in E\}$, it is possible to write a program P for an architecture that allows simultaneous accesses to at most d memory banks, so that P has G as its conflict graph

Proving the first assumption is fairly trivial: simulate the execution of P and have the simulator detect the object conflicts required for constructing G. The second assumption can be proven by giving an algorithm that generates the desired program:

```
for each edge e {
    1. generate code that requires degree(e)
       simultaneous memory accesses to the
       nodes connected by e;
    2. generate enough nop instructions to avoid
       interference with other instructions;
}

for each node n not appearing in any edge e {
    1. generate code that accesses n;
    2. generate enough nop instructions to avoid
       interference with other instructions;
}
```

3 DETERMINING THE ALLOCATION ORDER

To maximally exploit parallellism, our method first imposes an order on the conflicts and then allocates the objects that appear in the most important conflicts first. This notion of *importance* is based on a combination of 2 criteria: how often an object is *used* and how often it *conflicts* with other objects. In this section, we show how such an order can be derived automatically from an execution trace of P.

3.1 Gathering information by simulation

In [8] we presented a simple but general model of both the memory architecture as well as the pipelined execution of a DSP architecture. Given an execution trace of P (the sequence of instructions as they are handled during the executionof P), where all instruction operands are manifest (i.e. their values are known), relevant object use information is collected by one run of the the pipeline simulation.

3.2 The conflict graph

Recall that a *conflict* $c(o_1, \ldots, o_m)$ between m objects $o_1, \ldots, o_m$ is the need for simultaneous access of these objects during program execution. A *use* of an object is any access to that object, either `read` or `write`. To minimize the execution time of P, the objects need to be placed in the different memory banks in such a way that the cpu cycle penalties (the number of extra cpu cycles needed to access the objects due to their allocation) for both the conflicts and the uses are minimal. While going over the execution trace with the simulator, a data structure $G = (V, E)$ called the *conflict graph* is build. This graph keeps track of the number of conflicts, the objects involved in them as well as the overall number of uses of each object:

- a *node* $n \in V$ represents an object (its identifier and its size) and the number of times that object has been accessed without being part of a conflict.
- a labeled *edge* $e \in E$ represents the conflicts involving the nodes it connects. The label indicates how many such conflicts arise.

Note that an edge can connect more than 2 nodes. Representing a conflict between 3 nodes can also be done by 3 normal edges that connect the 3 possible pairs of nodes (see figure 1), but this results in a loss of knowledge. Indeed, using 3 edges that connect 2 nodes each, as is depicted in *(a)*, gives the impression that there are 3 possibly independent conflicts. The knowledge that it really is 1 conflict in which all nodes are accessed during the same cpu cycle is lost. Clearly, graph *(b)* has a notion of time that graph *(a)* has not: it can express that certain conflicts between pairs of objects happen simultaneously. We call edges that connect more than 2 nodes *multi-edges*. Note that *self-referencing edges* may exist: an edge can be connected to a certain node more than once so multiple accesses to the same object within a conflict can be represented in G.

It is clear that the dominating factors in the allocation are the number of times each object is used and the number of conflicts. If an object is used very often, it should reside in the fastest memory bank possible. If a number of objects are accessed simultaneously, they should be assigned to that set of memory banks that allow these accesses in the shortest possible time.

3.3 Deciding object allocation order

The algorithm used to do the allocation is based on the use of a set of tables $T = \{T_i \mid 1 \leq i \leq n\}$ where $n = max\{degree(e) \mid e \in E\}$. Each table T_i has $\binom{\#B + i - 1}{i}$ entries (i-combinations with repetition of $\#B$). Each entry in T_i is a function

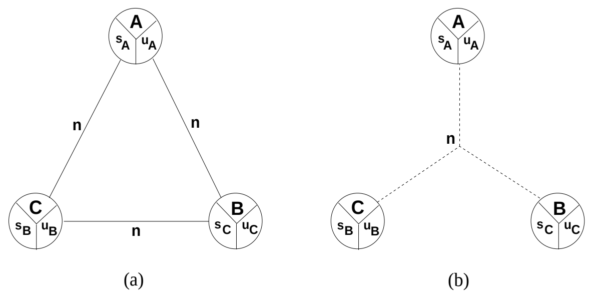

Figure 1: Edges connecting more than 2 nodes represent additional knowledge.

$$f(b_1, ..., b_i) : B \times B \times \ldots \times B \rightarrow \mathbb{N}$$

where $f(b_1, ..., b_i)$ is the number of cycles lost due to a conflict of degree i (i.e. a conflict involving i objects) where one object is in bank b_1, one object is in b_2, ... and one object is in b_i. Once both G and T are generated, we combine the information they contain into a new table T_{final} upon which our heuristic is built. For each edge[1] in G, there is an entry in T_{final} that indicates what loss of cpu cycles can be avoided by doing a good allocation of the objects involved.

The basic idea is to try to resolve the edge that represents the most costly conflicts first. This is based upon the assumption that a minimalization of the number of conflicts corresponds to a maximalization of the execution speed of the program, provided often used objects are assigned to the fastest banks. The following heuristic approximates the biggest possible gain in cpu cycles for an edge $e \in E$ that connects nodes $n_1, \ldots, n_m$ (with $m \leq degree(e)$: we take each node into account only once):

$$GAIN_e = WORST_e - BEST_e \tag{1}$$

where $n = degree(e)$ and

$$BEST_e = f(b'_1, \ldots, b'_n) \times conflicts(e) + \sum_{i=1}^{m} uses(n_i) \times (ws(b'_i) + 1) \tag{2}$$

$$WORST_e = f(b_{slow}, \ldots, b_{slow}) \times conflicts(e) + \sum_{i=1}^{m} uses(n_i) \times (ws(b_{slow}) + 1) \tag{3}$$

with $b'_1, \ldots, b'_n$ such that

$$f(b'_1, \ldots, b'_n) = \min_{b'_1, \ldots, b'_n} f(b'_1, \ldots, b'_n) \in T_n$$

$$\forall i, 1 \leq i \leq \#B : ws(b_{slow}) \geq ws(b_i)$$

$uses(n_i)$ is the number of times the object of node n_i was accessed without being part of the current conflict (i.e.

[1]Some nodes in G may not be connected to any other nodes. For our algorithm to work, we add a self-referencing edge to them labeled with 0 number of uses.

the conflict represented by e), $conflicts(e)$ is the number of conflicts of type e and b_{slow} is the slowest memory bank available. Note that we use $ws(b_i) + 1$ instead of just $ws(b_i)$ because in general the latter is 0 for the fastest banks, which would eliminate the influence of the number of uses besides those of the current conflict. The entries in T_{final} are ordered by decreasing value of the $GAIN_e$ field of the table. The objects of the edges with a higher $GAIN_e$ value will be allocated first.

4 TOWARDS AUTOMATIC ALLOCATION

4.1 The heuristic

Given T_{final} it is possible to automate the allocation process. The allocation guidelines handled by a prototype system we have developed include:

- We try to assign the most often used objects in nodes $n_1, \dots, n_{degree(e)}$ to the fastest memory banks in $b'_1, \dots, b'_{degree(e)}$.
- If multiple f_j have a minimal value, we try to spread the objects over as many different banks as possible.
- If objects have self-referencing conflicts, they are assigned faster banks which allow multiple accesses per cycle (if these exist).
- If there are multiple equally fast memory banks to assign an object to, the bank with the most free space left is picked.

The allocation is based on a single iteration over the entries in T_{final} in the order in which they were sorted. All entries start as unmarked. For each entry that is not marked, all nodes in it are allocated according to the guidelines above. This entry is then marked, as well as all entries that contain only nodes that have already been allocated.

4.2 A small example

Assume a target architecture with $B = \{b_1, b_2, b_3\}$ where the banks have sizes of 25, 25 and 1000 words respectively. Each cpu cycle, 2 accesses to b_1 are possible. b_1 and b_2 can be accessed in parallel, resulting in 1 access to b_1 and 1 access to b_2 during the same cycle. b_3 is slower, having a waitstate of 1. b_3 can not be accessed in parallel with either b_1 or b_2. This results in $T = \{T_1, T_2, T_3\}$ with T_3 shown in table 2. Instructions are of the form `mnem src1, src2, dst` where `src1` and `src2` can be an immediate value, a register or an address of/into an object and `dst` can be either a register or an address. Since immediates and registers are irrelevant for our concerns, they are replaced by a `-`. Accesses to an object are represented by `#x` where `x` is the name of the object. Each memory cell of any bank can hold exactly 1 instruction (or 1 word of data). The pipeline has 5 stages: stage 1 fetches the instruction, stage 2 decodes it and stage 3 reads the source operands `src1` and `src2` from memory. Stage 4 executes the instruction and finally during stage 5 the result is written to `dst`.

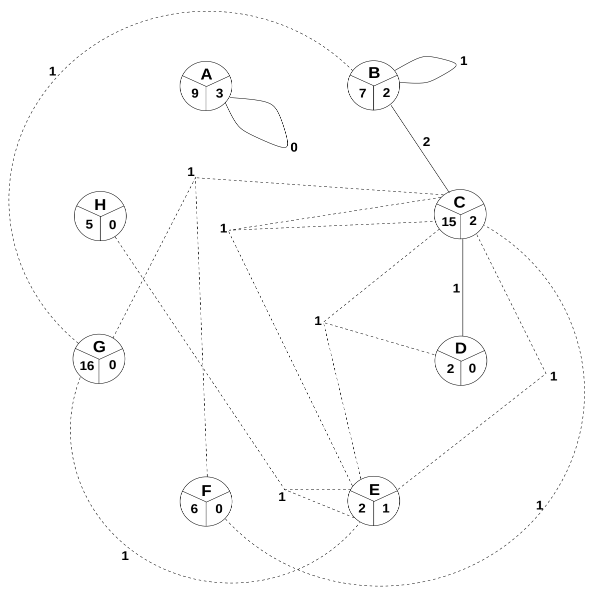

Figure 2: Conflict graph for the example.

Consider the execution trace in figure 3 featuring 8 objects **A** (size 9), **B** (size 7), **C** (size 15), **D** (size 2), **E** (size 2), **F** (size 6), **G** (size 16) and **H** (size 5). The conflict

```
01 mnem #A,  -,  -        13 mnem  -, #C, #E
02 mnem  -, #A,  -        14 mnem #F, #C, #E
03 mnem  -,  -, #A        15 mnem #C, #C, #C
04 mnem  -,  -,  -        16 mnem  -, #C, #F
05 mnem  -,  -,  -        17 mnem #B,  -, #B
06 mnem  -,  -,  -        18 mnem #G, #C, #E
07 mnem  -, #B,  -        19 mnem #G,  -, #E
08 mnem #B, #C, #D        20 mnem #E, #H,  -
09 mnem #D, #C, #C        21 mnem  -,  -,  -
10 mnem #D,  -, #B        22 mnem #E, #G, #C
11 mnem #D, #E,  -        23 mnem  -,  -, #B
12 mnem #B,  -,  -
```

Figure 3: Trace for the example.

graph for this trace is shown in figure 2. Note the edges that connect more than 2 nodes (dotted lines) as well as the self-referencing edges. From T and G, the values for $BEST_e$ and $WORST_e$ (and hence for $GAIN_e$) can be derived, resulting in T_{final} (see table 1).

According to the results from table 1, the object allocation is done as follows:

1. First try to assign **C** since it has 44 uses in total, which is more than both **D** (12 uses) and **E** (28 uses). Since there exist conflicts in which **C** is accessed twice during the same cpu cycle (in this example, there is only one such conflict: (C, C, E)), **C** is put in the only memory bank that allows this, i.e. b_1. Next, **E** is considered. It also has a conflict where it is accessed twice, so **E** is also put in b_1 since it still fits into that bank. **D** is never accessed more than once during the same conflict and gets put into b_2 since f(b_1,b_1,b_2) = f(b_1,b_1,b_1) = 1: there is no need to take up space in

edge	#	$WORST_e$	$BEST_e$	$GAIN_e$
(D,E,C)	1	41	19	22
(B,C)	2	34	14	20
(C,C,E)	1	35	16	19
(C,E)	1	35	16	19
(G,C,F)	1	31	14	17
(D,C)	1	27	12	15
(F,C)	1	25	11	14
(G,B)	1	19	8	11
(E,G)	1	19	8	11
(E,H,E)	1	15	6	9
(B,B)	1	13	5	8
(A,A)	0	6	3	3

Table 1: T_{final} for the example.

b_1 when allocating the object to b_2 doesn't make the handling of the conflict any slower. Entry (D,E,C) in T_{final} is marked as done. Also, entries (C,C,E), (C,E) and (D,E) are marked as done since all of the objects featured in any of these entries already have been allocated to a memory bank.

2. When considering entry (B,C), we only need to allocate **B** since **C** has a place already. **B** has an entry in T_{final} where it is accessed twice (i.e. (B,B)) so it is put in b_1. Both (B,C) and (B,B) are marked as done.
3. For entry (G,C,F), **G** and **F** still need to be allocated. **G** has more uses than **F** so **G** is put into b_2 while **F**, which no longer fits in either b_1 or b_2, is put into the slow bank b_3. Entries (G,C,F), (F,C), (G,B) and (E,G) are marked as done.
4. Entry (E,H,E) only has **H** as an object that still needs a place to reside during execution. H still fits into b_2 so it is put there. (E,H,E) is marked as done.
5. The only entry left is (A,A). **A** is put into the only memory bank that has enough space left for it: b_3. (A,A) is marked as done and the algorithm terminates.

The final allocation is represented in figure 4. In the worst case, which for this architecture means allocating all objects to b_3, the execution of the example would take 82 cpu cycles (we assume that an instruction is fetched from the I-cache each cycle and that all instructions take 1 cpu cycle to execute). Given the allocation above, this is reduced to 38 cycles. The total penalty for the conflicts due to the initial bad allocation has been reduced from 55 to 11.

5 RELATED WORK

Assigning static objects to memory banks is clearly something that should be incorporated into (or after) the linkage process, since unlike the compiler, the linker knows the memory map. Research that has focused on (post) link-time optimization (for general purpose processors) includes [16, 14, 13, 7, 2, 4, 3, 12].

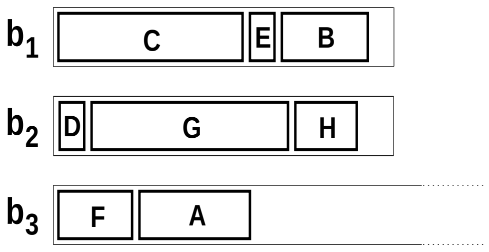

Figure 4: Final allocation for the example.

T_3	
f	*# lost cycles*
$f(b_1,b_1,b_1)$	1
$f(b_1,b_1,b_2)$	1
$f(b_1,b_2,b_2)$	1
$f(b_1,b_2,b_3)$	2
$f(b_1,b_1,b_3)$	2
$f(b_2,b_2,b_2)$	2
$f(b_2,b_2,b_3)$	3
$f(b_1,b_3,b_3)$	4
$f(b_2,b_3,b_3)$	4
$f(b_3,b_3,b_3)$	5

Table 2: Table of penalties for conflicts of degree 3 for the example.

A problem similar to the one in this paper is tackled in chapter 11 of [1]. Given P, O and a target cycle budget, a memory bank configuration (and the allocation of the objects to it) is derived, such that the power consumption versus execution speed trade-off is as good as possible. [11] minimizes power consumption for existing embedded DSP processors with only 2 memory banks which have the same characteristics.

[6] suggests a technique where a pre-compiler phase is used to eliminate objects that are not used by the program or to see which objects can be put at the same locations. This is orthogonal to what we do at link-time.

Other research such as [9] or [5] focuses on using the typical addressing methods of DSP processors to optimize data placement for speed, but they do not consider the different characteristics of the memory banks.

6 FUTURE WORK

Size does matter Equations 1, 2 and 3 don't take the size of objects into account. However, an object's size can play a major role in certain special cases. If the most often used object is so big that it occupies the entire fast memory space, the others will be allocated to the slow memory, although the total of their uses may be quite a bit bigger. On the other hand, we may still have some fast memory space left when all objects have been assigned to memory. A postprocessing step that duplicates objects and copies them into that space may improve parallellism. Another size related concept is *switch cost*: if an object is very heavily accessed but is too big to reside in a fast memory bank, it may still be worth it to add code that copies parts of the object to the fast bank as they are needed. Another option may be to split the object and allocate the parts individually. Most of these techniques require another run over the code to check which copy/part of an object should be accessed at which point in time. Moreover, this may also result in additional code if non read-only objects need to be kept consistent.

Lifetime analysis and dynamic objects It is clear that our algorithm can be improved by taking the life time of objects into account: objects with non-overlapping life times can be assigned to the same memory block. The fact that two objects have non-overlapping life times will be visible in the conflict graph by adding some[2] extra edges. When allocating an object, the space occupied by already allocated objects that are dead when that object is live can be reused. Once life times of objects are taken into account, and since we start from a manifest trace, dynamic objects allocated by `malloc()` can be placed optimally in memory using the same mechanism as for the static objects.

Implementation for a target architecture As the topics above focus on the relation between data placement and code, we will have to migrate our tool (which currently performs the algorithm presented on a parametrized model of a DSP) to a specific architecture, as code manipulation is needed there.

7 CONCLUSIONS

We have shown the automatic generation of a table that can guide the allocation of static objects to the memory banks of a DSP architecture. Instead of using a search algorithm that requires an evaluation function in the form of a program simulation that is executed multiple times, we extract the necessary conflict information from just one simulation. Using a conflict graph to represent that information, as well as tables that hold the architecture dependent allocation penalties, we are able to impose an order on the estimated cost of access conflicts. This order is the basis for automated allocation of objects, allocating the objects involved in the most costly conflicts first.

8 ACKNOWLEDGEMENTS

This work is sponsored by the Fund for Scientific Research - Flanders under grant 3G001998.

References

[1] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology.* Kluwer Academic Publishers, 1998.

[2] R. Cohn, D. Goodwin, P. Lowney, and N. Rubin. Spike: An Optimizer for Aplha/NT Executables. *USENIX Windows NT Workshop*, August 1997.

[3] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA*, pages 1013–1019, June 2000.

[4] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):378–415, March 2000.

[5] E. Eckstein and A. Krall. Minimizing Cost of Local Variables Access for DSP-processors. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems, Atlanta, GA USA*, pages 20–27, May 1999.

[6] P. Ellervee, M. Miranda, F. Catthoor, and A. Hemani. Exploiting Data Transfer Locality in Memory Mapping. In *Proceedings of the 25th IEEE Euromicro Conference, Milan, Italy*, pages 14–21, September 1999.

[7] M. F. Fernandez. *A Retargettable, Optimizing Linker.* PhD thesis, Princeton University, USA, January 1996.

[8] P. Keyngnaert, B. Demoen, B. De Sutter, and K. De Bosschere. Trace-based Memory Layout Optimization for DSPs. Technical Report CW282, K.U.Leuven, Belgium, March 2000.

[9] N. Kogure, N. Sugino, and A. Nishihara. Memory Allocation Method for Indirect Addressing DSPs with $\pm$ Update Operations. *IEICE TRANS. FUNDAMENTALS*, E81-A(3):420–428, March 1998.

[10] D. C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, USA, 1992.

[11] T. C. Lee and V. Tiwari. A Memory Allocation Technique for Low-Energy Embedded DSP Software. In *Proceedings of the 1995 IEEE Symposium on Low Power Electronics, San Diego, CA, USA*, October 1995.

[12] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. `alto`: A Link-Time Optimizer for the Compaq Alpha. *Software Practice and Experience*, 31:67–101, January 2001.

[13] A. Srivastava and D. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, pages 1–8, March 1993.

[14] A. Srivastava and D. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 49–60, June 1994.

[15] Texas Instruments' Visual Linker, see `http://dspvillage.ti.com/docs/ccstudio/`.

[16] D. W. Wall. Global Register Allocation at Link Time. In *Proceedings of the ACM SIGPLAN '86 Conference on Compiler Construction.*, pages 264–275, 1986.

[2]existing edges represent some of the life time overlaps already