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

January 7, 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 (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 linker 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.

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 don't; some banks allow more than one access per cpu cycle or have a smaller waitstate. For this kind of processing units, data placement has severe implications on the execution speed of an application. Objects that are often accessed 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 variables from their source code into a visual representation of the memory architecture they target (e.g. [19]), but it takes a lot of time and expertise to come up with a good placement. In this text, we present a novel way of automating this process.

In section 2 we define the problem and show it is NP complete. In section 3 we present a method to gather the relevant information for subsequent allocation of objects to memory banks. We show a heuristic to generate an ordered sequence by which the various objects should be allocated. In section 4, a first proposal is made for automatic allocation using the datastructures defined in section 3. Section 5 illustrates all this with an example. The final sections 6 and 7 consider related work and possible future work respectively.

2 The problem and it's complexity

2.1 The problem

Given are the set of static[1] objects O used by a program P, as well as the set of memory banks M of the target architecture. The goal is to find an allocation function $alloc : O \rightarrow M$ that minimizes the execution time of P.

2.2 Problem complexity

Consider the graph coloring problem[16], which has been extensively used in register allocation [4, 2]: given a graph, can its nodes be colored with n colors, provided that only nodes that are not connected by an edge can be assigned the same color? Now consider a memory architecture with n memory banks, all of which have waitstate 0 and allow for 1 access per cycle. Moreover, none of them can be accessed in parallel. Trying to assign the static objects of a program to these banks in such a way that no conflicts arise during program execution[2] is essentially the same problem. Since the first problem is known to be NP[13], the second one must also be NP. Moreover, the second problem is just a special case of the more general problem addressed in this document, where banks can have different waitstates and allow for parallel access. This prohibits the possibility of finding an optimal solution for non-trivial applications within a reasonable time. Instead, our goal is to find a good approximation of the optimal solution in an acceptable amount of time.

3 Defining the allocation order

Our allocation method treats the most important objects first. Within the context considered, this *importance* is based on 2 notions: how often an object is *used* and how often it *conflicts* with other objects. Clearly, objects that are used often and/or appear in a lot of access conflicts need to be allocated to the fastest memory banks. This implies the need for an order on the set of objects that indicates which objects have the most uses and constraints placed upon them and hence should be allocated first to maximally exploit parallellism. In this section, we show how such an order can automatically be derived from an execution trace of the program to be optimized.

3.1 Gathering information by simulation

In [11] we presented a simple but general model of both the memory architecture as well as the pipelined execution of a DSP architecture. An execution trace of a program is given as input to a simulator modeled as a finite state machine. By adding or removing states, the accuracy of the finite state machine simulation can be varied according to what is needed. The output of the simulation is the number of cpu cycles needed to execute the trace as well as details about the pipeline conflicts. In the execution trace, all arguments to the instructions are assumed manifest[3]. We ignore instruction cache details by simply assuming that it is there and delivers a new instruction each cpu cycle. No attempt is made to reschedule instructions. Instead, we assume that the compiler produces high quality code but that the assignment of datastructures to memory locations (banks in particular) is potentially very poor and hence can be improved upon. Also, objects

[1]see section 7 about future work for some thoughts on the handling of dynamic objects
[2]A conflict between objects is the need to access 2 or more objects during the same cpu cycle
[3]i.e. their values are known

are treated as indivisible objects, e.g. the elements of an array should be placed at consecutive memory addresses. Splitting up objects would also imply changing the code which is beyond our current aims. The interaction between data placement and code generation is among the topics for future research.

An optimal solution for the problem can be found by using the simulation as the evaluation function for an appropriate search strategy. However, repeatedly simulating a program is painstakingly slow. Such a search process can be sped up by either reducing the number of evaluations or reducing the cost of the evaluation. We opt for a variant of the former method by using the pipeline simulation just once to gather relevant information. A heuristic then uses this information to determine an order by which the objects should be allocated to the memory banks, and in which bank each object should reside. Since we only have one simulation, we can afford to use a detailed variant of the finite state machine simulator.

3.2 Conflict graph

We define a conflict $c(o_1, \ldots, o_m)$ between m objects $o_1, \ldots, o_m$ as the need for simultaneous access of these objects during program execution. The use of an object is any access of that object, either `read` or `write`. To minimize the execution time of the program, the objects need to be placed in the different memory banks in such a way that the cpu cycle penalties[4] for both the conflicts and the uses are minimal. While going over the execution trace with the finite state machine, a data structure called the *conflict graph* $G = (V, E)$ 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 $n \in V$ contains an object (its identifier and its size) and the number of times that object has been accessed without being part of a conflict.
- an edge $e \in E$ represents the conflicts involving the nodes connected by it. A label attached to the edge indicates how many such conflicts appear in the worst case.

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)*) would give the impression that there are 3 possibly independent conflicts. The fact 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, i.e. an edge can be connected to a certain node more than once. This takes into account multiple accesses to the same object within a conflict. In figure 2 *(c)*, node **B** has a self-referencing edge.

Assume 2 nodes A and B exist, taking part in 2 conflicts: $c_1(A, B, B)$, which occurs 10 times, and $c_2(B, B)$, which occurs only twice. Figure 2 illustrates 3 possibilities to model these conflicts. The way conflicts are modeled can have a severe impact on the quality of the allocation: if the simple heuristic of resolving the most frequently occuring conflict first is used, each graph may, under certain circumstances, lead to different results. In (a), the conflict between **A** and **B** is resolved first, but this may cause **B** to be allocated to a memory bank that doesn't allow 2 accesses during each cpu cycle, so the self-referencing edge of **B** can not be dealt with. In (b) we resolve the self-referencing conflict first, but duplicating nodes may make the decision problem much harder as information concerning one particular conflict is spread all over the graph. Graph (c) allows us to deal with both decisions concerning the placement of **A** and **B** at once. Therefore, allowing edges to have a degree[5] higher than 2 seems the best option.

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 several objects are accessed simultaneously, they should be assigned to (possibly different) memory banks that allow these accesses in the shortest possible time. The importance of the size factor is described in the section about future work. The reason we currently ignore this aspect is that either a large object is used very often and then it is allocated early in the process (to a fast bank, if it fits into it), or it is not used all that often in which case it doesn't really matter that it is only allocated to a slow memory bank later on.

[4]i.e. the number of extra cpu cycles needed to access the objects involved due to their allocation

[5]The degree of an edge is the number of nodes it connects

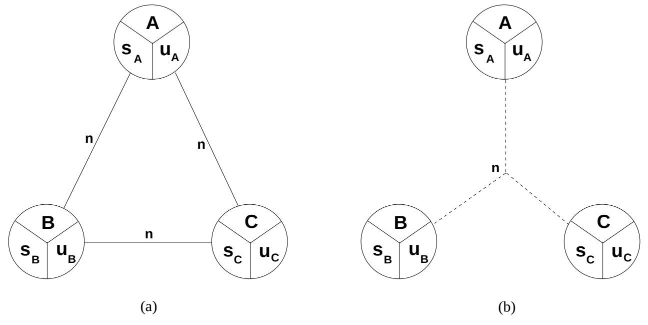

Figure 1: Edges connecting more than 2 nodes represent additional knowledge.

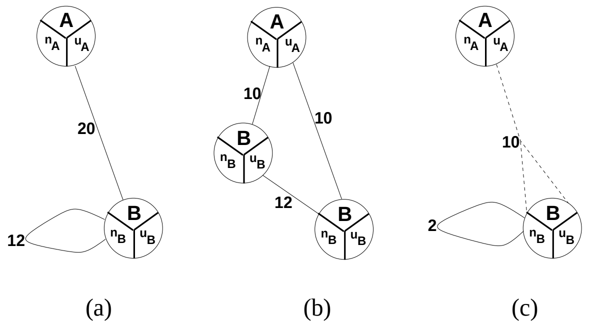

Figure 2: Edges connecting more than 2 nodes introduce additional benefits.

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 $\binom{\#Banks + i - 1}{i}$ entries[6] where $Banks$ is the set of memory banks in the architecture. Each entry in T_i is a function

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

where b_j $(j \in [1..i])$ is a memory bank and $f(b_1, ..., b_i)$ 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[7] in G, there is an entry in T_{final} that indicates the loss of cpu cycles that 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. We propose the following heuristic that approximates the biggest possible gain in cpu cycles for an edge $e \in E$ that connects nodes $N_1, \ldots, N_m$[8]:

$$GAIN_e = WORST_e - BEST_e$$

where $n = degree(e)$ and

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

[6] i-combinations with repetition of $\#Banks$

[7] 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.

[8] with $m \leq degree(e)$: we take each node into account only once.

$$WORST_e = f(b_{slow}, \ldots, b_{slow}) \times conflicts(e) \; + \; \sum_{i=1}^{m} uses(N_i) \times (waitstate(b_{slow}) \; + \; 1)$$

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

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

$$\forall i, 1 \leq i \leq \#Banks: \; waitstate(b_{slow}) \geq waitstate(b_i)$$

$uses(N_i)$ is the number of times the object of node N_i was accessed without being part of the current conflict (i.e. the conflict represented by e), $conflicts(e)$ is the number of conflicts of type e, b_{slow} is the slowest memory bank available and $waitstate(b_i)$ is the waitstate of bank b_i. Note that we use *waitstate() + 1* instead of just *waitstate()* because in general the latter is 0 for the fastest banks, which would eliminate the influence of the number of uses not part 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 should be allocated first.

4 Towards automatic allocation

T_{final} can be used to guide manual allocation by the programmer. However, given the information in the table, it should be possible to also have this final step in the allocation process automated. The allocation guidelines handled by a prototype system we have developed include:

- When assigning the objects of nodes $N_1, \ldots, N_{degree(e)}$ to banks $b'_1, \ldots, b'_{degree(e)}$, we try to assign the most often used objects to the fastest banks.
- If multiple f values in T_i are minimal, we try to spread the objects over as many different banks as possible.
- If nodes of objects have self-referencing edges in G or appear more than once as the node of a multi edge in G, they are assigned to faster banks which allow multiple accesses per cycle (if such banks exist).
- If one object needs to be assigned and there are multiple possibilities of equally fast memory banks, 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. Within such an entry, nodes with more uses are allocated first. This entry is then marked, as well as all entries that contain only nodes that have already been allocated. Currently, a post processing checks for memory banks that have leftover space as it may be beneficial to duplicate read only objects to maximize parallellism. The linker can then decide which version of the object needs to be accessed at which point in the code, and modify the addresses in the instructions accordingly. In the next section, we illustrate our approach with an example.

5 A small example

Assume a target architecture with $Banks = \{b_1, b_2, b_3\}$ 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 $T = \{T_1, T_2, T_3\}$ as shown in tables 1, 2 and 3.

T_1	
f	*# lost cycles*
$f(b_1)$	0
$f(b_2)$	0
$f(b_3)$	1

Table 1: Table of penalties for conflicts of degree 1 for the example.

T_2	
f	*# lost cycles*
$f(b_1, b_1)$	0
$f(b_1, b_2)$	0
$f(b_2, b_2)$	1
$f(b_1, b_3)$	2
$f(b_2, b_3)$	2
$f(b_3, b_3)$	3

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

Instructions are of the form `mnemonic 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`.

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).

```
instr #A,  -,  -
instr  -, #A,  -
instr  -,  -, #A
instr  -,  -,  -
instr  -,  -,  -
instr  -,  -,  -
instr  -, #B,  -
instr #B, #C, #D
instr #D, #C, #C
instr #D,  -, #B
instr #D, #E,  -
instr #B,  -,  -
instr  -, #C, #E
instr #F, #C, #E
instr #C, #C, #C
instr  -, #C, #F
instr #B,  -, #B
instr #G, #C, #E
instr #G,  -, #E
instr #E, #H,  -
instr  -,  -,  -
instr #E, #G, #C
instr  -,  -, #B
```

Figure 3: Trace for the example.

The conflict graph for this trace is shown in figure 4. Note the edges that connect more than 2 nodes (dotted lines) as well as the self-referencing edges of **A** and **B**.

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 4 where the entries are ordered according to decreasing $GAIN_e$ value).

According to the results from table 4, the object allocation is done as follows:

1. First try to assign **C** since is 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 place in 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

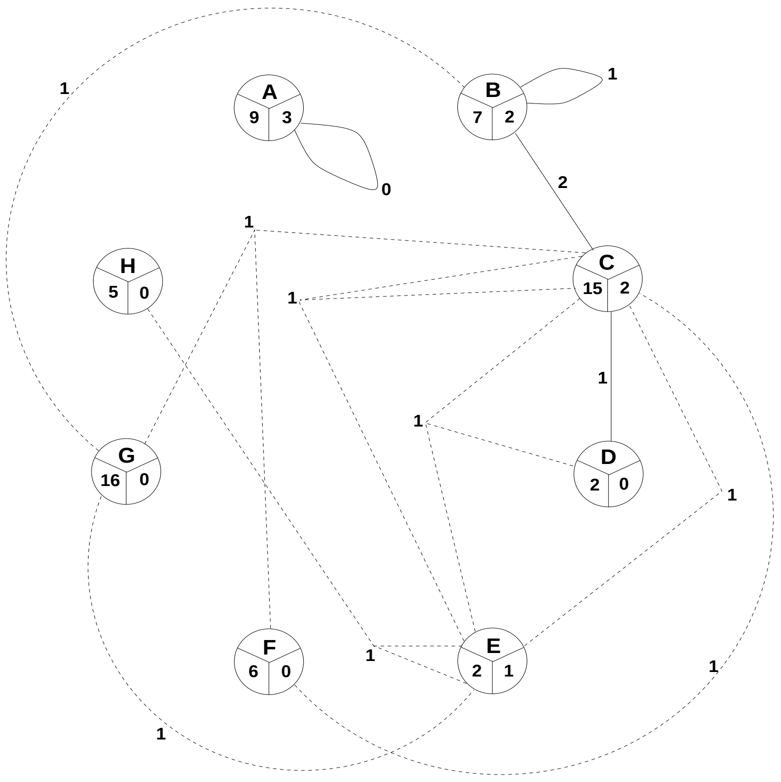

Figure 4: Conflict graph for the example.

edge	*conflicts*	$WORST_e$	$BEST_e$	$GAIN_e$
(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 4: T_{final}: heuristically deciding the order for conflict resolution for the example from section 2.6.

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 5. 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.

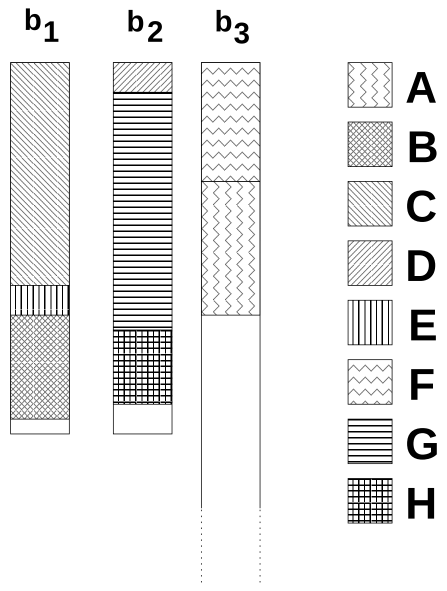

Figure 5: Final allocation for the example.

6 Related Work

A compiler [1] produces relocatable code that allows the linker [14] to put both data and code at the appropriate addresses in memory. Assigning 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 [20, 18, 17, 10, 5, 7, 6, 15].

A problem similar to the one in this paper (together with a methodology for solving it) is described in chapter 11 of [3]. Given a program and the objects used by it, 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.

[9] 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 layouted at the same locations. This is orthogonal to what we do at link-time.

Other research such as [12] or [8] focuses on using the typical addressing methods of DSP processors to optimize data placement, but they do not consider the memory hierarchy.

7 Future work

Since all instruction operands in the trace are manifest, we can find the dynamic objects of the program by looking for `malloc` (or similar) instruction sequences: we then know the address space each object occupies, as well as its size.

Size does matter 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 group of objects are accessed simultaneously, they should be assigned to (possibly different) memory banks that allow these accesses in the shortest possible time. However, the size of an object can play a major role which is illustrated in figure 6. The conflict edge between **A** and **B** is removed early on because this edge represents the highest number of conflicts. However, after the allocation of **A** and **B** to small but fast memory banks, all other objects need to be put into the large, slow memory because the sizes of the already allocated objects take up almost all of the space in the faster banks. Our algorithm needs to address the problems that are due to the size of objects better.

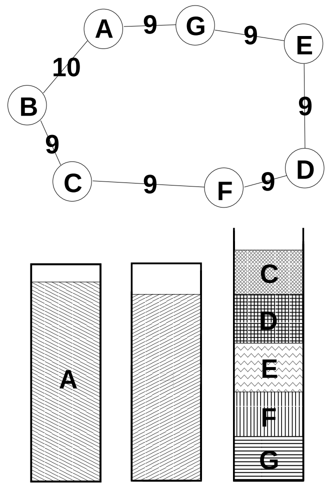

Figure 6: The importance of object sizes.

Size can also play a role the other way around, i.e. if we have space left when all objects have been assigned to memory. A simple way would be to have a postprocessing step that duplicates read only objects and copies them into the available space if this is beneficial for parallellism. Obviously, this requires another run over the linked code to check which copy of an object should be accessed at which point in time. However, this may also result in additional code if objects need to be kept consistent.

A third interesting look at object size may result in the concept of switch cost. Indeed, 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. This is related to the research about compiler controlled (smart) caches.

Finally, an object that is too big may also be splitted into separate objects that can then be alloated individually. Clearly, code may need to be added or changed here.

Dynamic objects Clearly, dynamic objects can be handled using the same algorithm as the static ones. We just have to know which dynamic objects there are, as well as their sizes. This is easily done while scanning the execution trace by looking for `malloc` sequences. The problem with dynamic objects is that they are only live for a certain (smaller) part of the program, so it would be highly beneficial to have some of them, that are not simultaneously live, occupy the same memory space. This requires 2 things we do not have at this moment: liveness information in the conflict graph and the possibility of adding code (as copying objects from one memory to another is needed then).

Implementation for a target architecture Once the results of our general prototype seem satisfactory, a transit should be made to a target architecture because all the intended refinements and/or extensions consider either adding or changing code. Also, for some instruction sets the order of the operands can influence the execution speed, so it is highly desirable that we take this effect into account for such systems.

8 Acknowledgments

This work is sponsored by the Fund for Scientific Research - Flanders under grant 3G001998. Personal correspondence with Saumya Debray of the University of Arizona, members of the ACCA research group at K.U.Leuven and members of the SEMP research group at IMEC was much appreciated.

References

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

[2] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, TX, USA, 1992.

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

[4] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markenstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[5] R. Cohn, D. Goodwin, P. Lowney, and N. Rubin. Spike: An optimizer for aplha/nt executables. *USENIX Windows NT Workshop*, August 1997.

[6] 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.

[7] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compression. *ACM Transactions on Programming Languages and Systems (TOPLAS).* To appear.

[8] 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.

[9] 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.

[10] M. F. Fernandez. *A Retargettable, optimizing linker.* PhD thesis, Princeton University, USA, January 1996.

[11] 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. Avaliable from `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW282.ps.gz`.

[12] N. Kogure, N. Sugino, and A. Nishihara. Memory allocation method for indirect addressing dsps with $\pm$ update operations. *IEICE TRANS. FUNDAMENTALS*, (3), March 1998.

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

[14] J. R. Levine, editor. *Linkers & Loaders.* Morgan Kaufman Publishers, San Francisco, California, USA, 2000.

[15] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. `alto`: A link-time optimizer for the compaq alpha. *Software Practice and Experience.* To appear (also available as Technical Report 98-14, Dec. 1998; this is a revised and updated version of Technical Report 96-15, Dept. of Computer Science, The University of Arizona, Tucson, USA).

[16] R. Nelson and R. J. Wilson, editors. *Graph Colourings (Proceedings of a Conference on Graph Colourings, Milton Keynes, 1988).* Longman Scientific & Technical, Essex, 1990.

[17] A. Srivastava and D. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, pages 1–8, March 1993.

[18] 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.

[19] Texas instruments' visual linker, see `http://dspvillage.ti.com/docs/ccstudio/`.

[20] D. W. Wall. Global register allocation at link time. In *Proceedings of the ACM SIGPLAN '86 Conference on Compiler Construction.*, pages 264–275, 1986.