

Trace-Based Memory Layout Optimization for DPSs

Peter Keyngnaert

Bart Demoen

Bjorn De Sutter

Koen De Bosschere

Report CW 282, March 2000



Katholieke Universiteit Leuven

Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Trace-Based Memory Layout Optimization for DSPs

Peter Keyngnaert

Bart Demoen

Bjorn De Sutter

Koen De Bosschere

Report CW 282, March 2000

Department of Computer Science, K.U.Leuven

Abstract

DSP processors often have a complex memory architecture, as their memory space, while addressed in a continuous way, physically consists of a number of different memory banks, each with their own access constraints and access times. Therefore, data placement has a major influence on the execution speed of a DSP application. We describe a mathematical model that allows us to address the problem at hand in a formal way. The input for this model is an execution trace of the program to optimize. We also show a number of solutions for the problem, based upon the model presented. First, an optimal but slow solution using a *generate and test* algorithm is presented. Next, we show how this solution can be sped up with only a minor loss of optimality by applying a *genetic algorithm*. Finally, we present a different but very fast solution, based upon the same model but using heuristics. An example for a simple, theoretical machine shows that the latter technique is very promising.

Trace-Based Memory Layout Optimization for DSPs

Peter Keyngnaert Bart Demoen

Bjorn De Sutter 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,koen.debosschere}@elis.rug.ac.be

Abstract

DSP processors often have a complex memory architecture, as their memory space, while addressed in a continuous way, physically consists of a number of different memory banks, each with their own access constraints and access times. Therefor, data placement has a major influence on the execution speed of a DSP application. We describe a mathematical model that allows us to address the problem at hand in a formal way. The input for this model is an execution trace of the program to optimize. We also show a number of solutions for the problem, based upon the model presented. First, an optimal but slow solution using a *generate and test* algorithm is presented. Next, we show how this solution can be sped up with only a minor loss of optimality by applying a *genetic algorithm*. Finally, we present a different but very fast solution, based upon the same model but using heuristics. An example for a simple, theoretical machine shows that the latter technique is very promising.

1 Introduction

1.1 The Problem

Traditionally, compilers optimizing for speed put all their efforts on the selection and scheduling of instructions. The data these instructions operate on does, for general purpose processors, not influence execution speed since memory is just one continuous address space where all memory locations are of the same type¹ (various levels of data and instruction caches exist, but there is no way of explicitly manipulating what's in them at a certain point in time²). However, this view of optimizing is not sufficient for DSPs since these processors have to access different types of memory where some accesses can even be dependent on others due to constraints on the memory: one part of memory may be faster than another and there may even be inter-memory constraints with respect to parallel data access. It is therefor intriguing that DSP compilers/linkers seem to ignore the problem of placing data in such a way that maximum parallelism (and hence maximum execution speed)

¹i.e. have the same physical characteristics

²In a limited way, this claim is not entirely correct: some architectures support such things as *cache freezing* or *software managed caching*

can be achieved. Indeed, DSP programmers still have to optimize their code by hand and have to manually indicate where certain data is placed. We call this the *memory layout problem* and in this report we make an attempt to tackle it. The reasons for doing this are twofold: first, the better the compiler generated code is, the less work the programmers have to do. Secondly, if the generated code is faster, a slower (and cheaper) DSP might be able to do the work instead of a faster (and more expensive) one. In both cases, the benefits are reflected in a decrease of costs for the DSP based industry. Optimizing by hand and/or writing the critical parts of the code in assembly instead of C is still more the case than the exception in the DSP industry (e.g. the development methodology Texas Instruments proposes for it's TMS320 series in [Texas Instruments, 99] explicitly mentions manually optimizing the assembly code generated by their C compiler), so solving the problem at hand is certainly worthwhile. The reason that our approach uses (an execution trace of) an executable program as it's input, is that the *memory map* (i.e. a formal description of how memory is organized on the architecture on which the program should run) is known to the linker but not to the compiler (e.g. in [Texas Instruments, 97]):

“The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes) or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces.”

1.2 A Small Example

A small example should illustrate the problem at hand. Consider a simplified DSP architecture with 2 memory banks, one of which, B_1 , can be accessed once during each 2 cpu cycles (so it's a 1-waitstate memory) and one of which, B_2 , can be accessed twice during 1 cpu cycle. Furthermore, assume there are no caches and there's a 4-stage pipeline (where the stages are, in this order: **F**etch, **D**ecode, **R**ead, **E**xecute). Also given is the following piece of code we wish to execute on this architecture:

```

...
neg    reg4
neg    reg2
add    src_addr2,src_addr3,dst_reg3
store  src_reg1,dst_addr1
...

```

It's immediately obvious that the code will have to be placed in B_2 (if memory size allows this and if there is no instruction cache available) since otherwise one cpu cycle is lost for every fetch and (code-fetching) performance would be halved³. But where to place the data? Placing both `src_addr2` and `src_addr3` in B_1 is inferior to placing one of them in B_1 and the other in B_2 . However, since the code is already residing in B_2 there may not be enough space left for all the data. A careful choice would then have to be made as for which data is important enough to reside

³This problem can be abstracted away though, by assuming that there is an instruction cache that ensures the fetching of 1 instruction each cycle.

in B_2 and which data gets placed in B_1 . A good starting heuristic would be to place the data that is used most frequently (i.e. in the innermost loops) in the fastest memory. In a cpu with more banks, additional constraints may exist due to the bus structure. All this indicates that a careful placement of data in memory may result in faster code execution.

1.3 Data vs Code

Data and code are closely intertwined. Given an executable program (and thus a given allocation scheme for all data), just fiddling with the memory addresses to reallocate the data in a more optimal way can lead to incorrectness. Consider e.g. an array that is accessed in a loop by adding an index to an offset: if the optimal data placement would be to have part of the array reside in one bank and the remaining part in another bank, the code needs to be changed accordingly (possibly needing two loops, one to iterate over the first part of the array and one to iterate over the other part). Changing the code to reflect the data reallocation adds another level of complexity to the problem and is a possible topic for future research. In this report, only reallocations that don't result in a need for changing the code are considered.

1.4 Hardware vs Software

Why not use additional cache memories instead of doing this software-based optimization? There are several (primarily economical) reasons that show the software solution to be more interesting:

- Extra cache memory increases the cost of the overall product. For the software, you pay once. For hardware, this is not the case as you pay the extra bit for each processor or board that rolls off your production line. If your company is producing for a multiple billion consumer market, 1 dollar extra per item is an enormous amount of money.
- Extra cache memory means extra power consumption. Having as low a power consumption as possible is vital for the DSP world. E.g., take a look at the booming market for GSM cellular phones: an additional power consumption of 10% means that the battery life will be cut down by 10% also. A lot of effort is done already to keep power consumption to a minimum, see e.g. [Catthoor, 98].
- What memory banks exactly will each cache be attached to? Giving each bank a separate cache doesn't solve the problem, it just moves it down one level onto cache memory allocation. This can be solved by using very fast caches with multiple accesses per cpu cycle and high inter-cache-access parallelism, but then the extra cost will be very high. Adding one cache in front of multiple banks also implies the need for very fast (and hence very expensive) cache memory.

2 A Mathematical Model of the Problem

2.1 A first approximated specification of the problem

We start from an execution trace of a given compiled program, all arguments to the instructions are assumed manifest⁴. We will ignore details of the instruction cache and we will also make no attempt to reschedule the instructions. These concerns can be taken into account at a later point:

⁴i.e. their values are known

the rationale is that usually the generated code has a high quality already, but the assignment of data structures to memory locations (banks in particular) is poor and can be improved upon.

The objects and the memory locations We will model memory locations and the operands of instructions – which we name **objects** – separately: the objects need to be placed somewhere in memory. Objects come from the set Obj . Objects have a size that is given by the function

$$osize : Obj \rightarrow \mathbb{N}_0$$

Memory locations are modeled as natural numbers $\in AS \subseteq \mathbb{N}$, which intuitively correspond to their effective addresses. Of course, object sizes have some restrictions with respect to the number of elements of AS .

The memory banks There are k memory banks B_i , $i = 1 \dots k$. Each bank is a subset of \mathbb{N} . The capacity of a bank B_i is just $\#B_i$. The union of all banks is denoted by $Banks = \bigcup_{i=1}^k \{b_i\}$. Note that interleaved⁵ banks are modeled by their contents, that $\forall b_i, b_j \in Banks : b_i \cap b_j = \emptyset$ and $\bigcup_{i=1}^k b_i = AS$.

The instructions The execution trace T is given as a series of instructions $instr_i$, $i = 1 \dots I$. For each instruction, it is known which objects it uses as operands, by the function ops whose result is a n -tuple of objects (n can vary per instruction): it should not be a (multi-)set, as in some architectures the order of the operands can influence the time an instruction takes to execute.

Assignment of objects to memory banks An assignment of objects to memory banks is a function

$$ass : Obj \rightarrow AS$$

An assignment has to obey the following constraints:

$$\forall a, b \in Obj : a \neq b \Rightarrow [ass(a), ass(a) + osize(a)[\cap [ass(b), ass(b) + osize(b)[= \emptyset$$

We will introduce some other constraints on the function ass later.

ass can be extended in a trivial way from an assignment of objects to an assignment of tuples of objects.

Note that one can relax ass to be a relation instead of a function. The constraints as defined above imply that there is no overlap between objects but actually there can be, i.e. if the objects are *live* during non-overlapping periods when executing the program. However, this would require adding some extra code for moving objects from one bank to another, something we're not considering at this moment (see the part about *costs* later on).

⁵In an interleaved memory architecture, consecutive addresses reside in different banks

Neighboring objects Very often objects are grouped together as arrays, which means that these objects must be assigned neighboring addresses - at least, if the code dictates so by e.g. using address register increments to go through the array. As such, it is clear that there does not need to be an explicit relation between arrays and *osize*. As we do not deal with addresses directly, we will model neighboring objects by assuming the existence of a function

$$next_to : Obj \times Obj \rightarrow Boolean$$

which captures exactly that. Given the function *next_to*, any assignment *ass* must obey the restriction:

$$\forall a, b \in Obj : next_to(a, b) \Rightarrow ass(b) = ass(a) + osize(a)$$

The function *next_to* itself must obey:

$$\forall a, b, x, y \in Obj : next_to(a, b) \ \& \ (x, y) \neq (b, a) \Rightarrow \neg next_to(a, y) \ \& \ \neg next_to(x, b).$$

The cost of an instruction We make the approximation that the cost of an instruction can be computed in isolation. Then the cost can be described as a function *cost* from an n-tuple of memory banks to \mathbb{R}^6 . There are in principle no restrictions on this cost function, but realistically, the cost of n-tuples for $4 \leq n$ is never needed as instructions have usually less than 4 operands in memory.

The cost of an execution trace with respect to an assignment For given sets *Banks* and *Obj* and a given function *next_to*, the cost of an execution trace *instr_k* with respect to an assignment *ass* can now be described by

$$Cost(ass) = \sum_{k=1}^I cost(ass(ops(instr_k)))$$

A first optimization problem consists now in finding an assignment *ass* that minimizes the *Cost(ass)*.

A second optimization problem In principle, assignments can vary during the execution of a program. E.g. if the program consists of two subprograms, that are executed after each other only once, and they work on different objects, then it could be beneficial to swap the sets of objects before switching from one subprogram to the other. In general, we will therefor consider the cost of a sequence of assignments $\{ass_i\}$, for $i = 1 \dots N$ as follows:

$$Cost(\{ass_i\}) = \sum_{k=1}^N \sum_{j=1}^I cost(ass_k(ops(instr_j))) + \sum_{k=1}^{N-1} switch_cost(ass_k, ass_{k+1})$$

⁶the nature of the instruction itself is completely ignored - maybe that is not good; also what is load and what is store is ignored. Would something like $Cost(ass) = \sum_{k=1}^I cost(ass(ops(instr_k)), instr_k)$ solve this?

where $switch_cost(a, b)$ represents the cost of switching from assignment a to assignment b . It is clear that

$$\forall assignments\ a : switch_cost(a, a) == 0.$$

Now the new optimization problem consists of finding a sequence of assignments ass_i , so that $Cost(\{ass_i\})$ is minimal.

A derived optimization problem Switching from an assignment a to an assignment b can be implemented in several ways. It is clear that one wants the implementation which minimizes $switch_cost(a, b)$. This problem seems easier than the memory layout problem, but is not entirely trivial, as an optimal implementation will depend on the instruction set, the characteristics of the banks, the available registers, the occupancy of the banks at the moment of the switch ...

The complexity of the first optimization problem The k -colorability problem is a problem that is known to be NP -complete (e.g. [Kozen, 92]). Given a graph $G = (V, E)$ (V is the set of vertices, E the set of edges) and an integer k , is there a coloring of G with k or fewer colors? A coloring is a map $\chi : V \rightarrow C$ such that no 2 adjacent vertices have the same color; i.e., if $(u, v) \in E$ then $\chi(u) \neq \chi(v)$. Using the concept of *reduction*⁷, we can prove that the memory layout problem is reducible to the k -coloring problem indicating the former is NP -complete. The reduction is straightforward by associating the nodes of the *interference graph* with the objects we want to put in memory. If there's an edge between 2 nodes in the *interference graph*, we say that the 2 objects corresponding with these nodes must reside in different⁸ memory banks.

2.2 A more realistic specification of the problem

A more refined cost function will take into account the pipeline: we will have to model time for doing that. We define a pipeline P as a series of stages s_i , $i = 1 \dots S$. Each instruction enters the pipeline at stage s_1 and leaves it when it has completed stage s_S or is flushed. An injective priority function $prio$, which gives each stage a unique priority number, defines a total order over the stages as follows:

$$prio : P \rightarrow \mathbb{N}$$

where

$$\forall s_i, s_j \in P : s_i \neq s_j \Leftrightarrow prio(s_i) \neq prio(s_j)$$

Priorities are needed as tie-breakers: when a conflict in the pipeline occurs, i.e. when two or more stages are fighting over the same resource, one of them must have priority over the others and use the resource. We will assume that a stage gets priority over another stage if its priority number is lower.

⁷A problem A is said to be *reducible* to a problem B if there is a way to encode instances x of problem A as instances $\sigma(x)$ of problem B . The encoding function σ is called a *reduction*.

⁸For this to be entirely accurate, we must consider a bank b_i with $ac(b_i) \geq 2$ to be $ac(b_i)$ different banks. A better way to formulate interference is to say that 2 nodes interfere if they need to be accessed during the same cpu cycle.

To determine which resources are needed by which stages at a certain point in time, we indicate which operands of an instruction are used by a certain stage by defining a function $sops$ (analogous to ops) which returns an n -tuple of objects:

$$\forall instr_i \in T, s_j \in P : sops(instr_i, s_j) = \{o \in Obj \mid instr_i \text{ accesses } o \text{ in } s_j\}$$

where obviously $sops(instr_i, s_j) \subseteq ops(instr_i)$ and $\bigcup_j sops(instr_i, s_j) = ops(instr_i)$.

In principle, each stage of the pipeline should complete it's operation on a certain instruction in 1 cpu cycle, so as to give us the desired execution time of 1 cycle for each instruction. However, memory architecture and memory layout imply certain conditions for which a stage can not finish it's job in 1 cycle. We define two functions ac and ca over $Banks$, which respectively give the maximum number of accesses per cycle and the number of cycles per access of a bank b_k :

$$ac : \bigcup_i \{b_i\} \rightarrow \mathbb{N}_0 \quad \text{and} \quad ca : \bigcup_i \{b_i\} \rightarrow \mathbb{N}_0$$

ac and ca allow us to model some of the restrictions the memory architecture poses on the access of banks:

- a bank b_k has a maximum of $ac(b_k)$ accesses per cycle: all extra accesses will have to wait
- a bank b_k can have a waitstate of w , which means that one access takes $ca(b_k) = 1 + w$ cycles: other stages of P may have to wait for the result of this access

There is another architectural restriction which is vital with respect to our problem: the bus structure. As the cpu is connected to memory via a number of buses, we have to consider these as well: the way the buses are connected as well as the throughput they can deliver may further constrain the model. E.g. if we have 2 banks b_0 and b_1 that both allow 2 accesses per cycle, but they are connected to the cpu by the same bus which only supports 2 transfers per cycle, we have a maximum of 2 accesses per cycle, even if memory would allow up to 4. We will therefor model buses bu_k as pairs (n_k, M_k) where

n_k is the maximum number of transfers per cycle for bus bu_k

and

$$M_k = \{b_i \mid bu_k \text{ connects } b_i \text{ to the cpu}\}$$

Intuitively, one would model P as a window W_P of size S sliding over T , i.e. as a snapshot of P at a certain point in time. However, using T as defined before would lead to a model that can not take into account all possible conflicts that may occur in P . This is illustrated by the following example where $Bcond$ is a *conditional branch*:

```

...
mnem1 op1_1,op1_2
mnem2 op2_1,op2_2
mnem3 op3_1,op3_2

```

```

Bcond cond,label
mnem4 op4_1,op4_2
mnem5 op5_1,op5_2
mnem6 op6_1,op6_2
mnem7 op7_1,op7_2
label: mnem8 op8_1,op8_2
...

```

If we e.g. have a 4-stage pipeline (**F**etch, **D**ecode, **R**ead and **E**xecute) and instruction **mnem6** is fetched from the same memory bank b_i as the operands of instruction **mnem4**, a problem arises if only 2 accesses per cycle are possible from b_i . If T is the execution trace (i.e. T is the series of instructions as they are executed by the processor), we will miss the fact that there is a pipeline conflict here since **mnem4** and **mnem6** are not in T and we can not flush the pipeline when **Bcond** is in the decode stage since we don't know whether the condition for branching is true or false. To avoid incomplete execution of instructions following a branch, the pipeline is flushed. However, by the time it is known that we are dealing with a branch instruction, another instruction has been fetched. This problem can not be simply ignored, as a branch often marks the end of a loop and as such is potentially executed a lot of times. A possible solution for this problem is to replace the execution trace T by a trace T_{ep} which is the series of instructions *of the program* (so instructions inserted by the pipeline mechanism are not included in T_{ep}) that consecutively enter the pipeline. For the above example, $T_{ep} = \{mnem1, mnem2, mnem3, Bcond, mnem4, mnem5, mnem6, mnem8\}$.

Pipeline				
time	Fetch	Decode	Read	Execute
t + 0	mnem1	-	-	-
t + 1	mnem2	mnem1	-	-
t + 2	mnem3	mnem2	mnem1	-
t + 3	Bcond	mnem3	mnem2	mnem1
t + 4	mnem4	Bcond	mnem3	mnem2
t + 5	mnem5	mnem4	Bcond	mnem1
t + 6	mnem6	mnem5	mnem4	Bcond
t + 7	mnem8	-	-	-

We can now define the mechanics of P as a transformation τ :

$$\tau(i_{s_1}, \dots, i_{s_S}, T_{ep}, state, duration)$$

and under *normal* conditions τ behaves as follows:

$$\tau(i_{s_1}, i_{s_2}, \dots, i_{s_{S-1}}, i_{s_S}, T_{ep}, i_{s_1} + 1, nml, 0) = (i_{s_1} + 1, i_{s_1}, \dots, i_{s_{S-2}}, i_{s_{S-1}}, T_{ep}, i_{s_1} + 2, nml, 0)$$

which basically is the aforementioned window sliding over the trace where i_{s_i} is the index in T_{ep} of the instruction that is currently in pipeline stage s_i .

Now, depending on how the pipeline was implemented, a lot of extra rules can refine the transformation. We will use the aforementioned 4-stage pipeline (modified by adding a fifth stage (**Writeback**)) to illustrate some of these rules. In general, the current state of this pipeline is given as a 9-tuple

$$(i_F, i_D, i_R, i_E, i_W, T_{ep}, i_{T_{ep}}, state, duration)$$

where

- i_x = the index (in T_{ep}) of the instruction in stage x , e.g. i_R is the index for the read stage
- T_{ep} is the trace as previously defined
- $i_{T_{ep}}$ is the index (in T_{ep}) of the next instruction to fetch when the pipeline is operating under *normal* conditions
- *state* indicates whether or not the pipeline is handling a special case (e.g. $state = nml$ when the pipeline operates under normal conditions, $state = usb$ when the pipeline is busy handling an undelayed standard branch, etc...)
- *duration* indicates how long the pipeline will continue to be in the current state (when $duration = 0$, it'll switch back to *normal* state)

Let's have a look at some examples now:

- normal operation:

$$\tau(i_{s_1}, i_{s_2}, i_{s_3}, i_{s_4}, i_{s_5}, T_{ep}, i_{s_1} + 1, nml, 0) = (i_{s_1} + 1, i_{s_1}, i_{s_2}, i_{s_3}, i_{s_4}, T_{ep}, i_{s_1} + 2, nml, 0)$$

- unconditional standard (= *nondelayed*) branch **BRA**:

$$\tau(i_{s_1}, i_{BRA}, i_{s_3}, i_{s_4}, i_{s_5}, T_{ep}, i_{s_1} + 1, nml, 0) = (i_{bub}, i_{bub}, i_{BRA}, i_{s_3}, i_{s_4}, T_{ep}, i_{s_1} + 1, usb, 0)$$

$$\tau(i_{bub}, i_{bub}, i_{BRA}, i_{s_3}, i_{s_4}, T_{ep}, i_{s_1} + 1, usb, 0) = (i_{bub}, i_{bub}, i_{bub}, i_{BRA}, i_{s_3}, T_{ep}, i_{s_1} + 1, nml, 0)$$

- program wait due to multicycle access (here: $i_{s_1} + 1$ is fetched from 2-waitstate memory so $ca(ass(T_{ep}[i_{s_1} + 1])) = 3$):

$$\tau(i_{s_1}, i_{s_2}, i_{s_3}, i_{s_4}, i_{s_5}, T_{ep}, i_{s_1} + 1, nml, 0) = (i_{s_1} + 1, i_{s_1}, i_{s_2}, i_{s_3}, i_{s_4}, T_{ep}, i_{s_1} + 2, mcf, 1)$$

$$\tau(i_{s_1} + 1, i_{s_1}, i_{s_2}, i_{s_3}, i_{s_4}, T_{ep}, i_{s_1} + 2, mcf, 1) = (i_{s_1} + 1, i_{bub}, i_{s_1}, i_{s_2}, i_{s_3}, T_{ep}, i_{s_1} + 2, mcf, 0)$$

$$\tau(i_{s_1} + 1, i_{bub}, i_{s_1}, i_{s_2}, i_{s_3}, T_{ep}, i_{s_1} + 2, mcf, 0) = (i_{s_1} + 1, i_{bub}, i_{bub}, i_{s_1}, i_{s_2}, T_{ep}, i_{s_1} + 2, nml, 0)$$

What we have defined here is actually a *finite state machine*. Note that the last 2 parameters (*state* and *duration*) can be merged together into one parameter (i.e. the cross product of their value sets can be used) in those cases where the cross product set is not too big, which is a realistic assumption since memory with 2 waitstates or more is almost never used.

The cost function can now be defined in a very simple way, i.e. it's the number of times τ has to be applied to the initial T_{ep} to get to a final state

$$(i_{\text{something}}, i_{\text{something}}, index_{\text{something}}, i_{\text{something}}, i_{\text{something}}, T_{ep}, i_{T_{ep}}, normal, 0)$$

where $i_{T_{ep}}$ is invalid (e.g. $i_{T_{ep}} > \#T_{ep}$). This is obvious since each application of the τ transformation corresponds to one cpu cycle of program execution. A more efficient cost function (in terms of evaluation cost) is to count the number of pipeline conflicts detected. Since we don't fiddle with the instructions in this first step, the number of pipeline conflicts is a valid indication of how good the current memory layout is: less conflicts means less *pipeline bubbles* (i_{bub}) or *pipeline stalls*, i.e. faster execution of the program.

3 Solutions Built upon the Model

3.1 An Optimal Solution

If $\#Banks = N_B$ and N_V is the number of variables used in T_{ep} , then we define a string $S = [s_1, \dots, s_{N_V}]$ where $s_i \in [1..N_B]$. Semantically, $s_i = c$ means that the i_{th} variable (we just assign a number to every variable so they are ordered) of T_{ep} resides in bank b_c . So now we have to find a string S_{opt} for which τ is minimal. We do this by generating all valid strings⁹ and then evaluating τ for each of them. We then just take the allocation for which the corresponding string resulted in the smallest τ value. It is clear that since evaluating τ is equivalent with simulating the program we wish to optimize, this solution is not viable in practice due to time constraints¹⁰.

- $\tau_{best} \leftarrow \infty$
- **while** (further valid strings can be generated) **do**
 - generate a new valid string s_{new}
 - $\tau_{new} \leftarrow \tau$ applied to s_{new}
 - **if** ($\tau_{new} < \tau_{best}$) **then** $\tau_{best} \leftarrow \tau_{new}$
- **return** τ_{best}

3.2 A Close-to-Optimal Solution

3.2.1 The Basic Genetic Algorithm

A more clever version of the previous generate-and-test algorithm can be built by using a genetic algorithm. The idea is that in the previous solution we new strings are generated blindly while

⁹Not all strings are valid. If we generated all strings that are possible, we might end up with a lot of allocations where the total size of all the objects assigned to a certain bank b_k is too large to fit into b_k .

¹⁰As an illustration of this, if we consider a program with just 50 objects which has to run on a DSP processor with 4 memory banks, we already have a worst case of having to simulate our program 4^{50} times.

this may be done more intelligently by accepting the fact that *good* strings will probably not differ that much from the optimal string (and vice versa). Starting from a number of random strings, the idea is to keep the fittest strings (i.e. those with the lowest τ value) and *breed* a new population from them. Fitter strings should be obtained with each iteration until a string is created that is fit enough to meet our requirements. One generic version of such a genetic algorithm which can be tuned to the needs at hand (from [Mitchell, 97]) is given here:

$GA(Fitness, Fitness_{threshold}, p, r, m)$

- **Initialize** population: $P \leftarrow$ Generate p hypotheses at random
- **Evaluate**: for each h in P , compute $Fitness(h)$
- **While** $[maxFitness(h)] < Fitness_{threshold}$ **do** Create a new generation P_S :
 1. Select: probabilistically select $(1 - r)p$ members of P to add to P_S . The probability $Pr(h_i)$ of selecting hypothesis h_i from P is given by $Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^p Fitness(h_j)}$
 2. Crossover: probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P , according to $Pr(h_i)$ given above. For each pair $\langle h_1, h_2 \rangle$ produce 2 offspring by applying the Crossover operator. Add all offspring to P_S .
 3. Mutate: choose m percent of the members of P_S with uniform probability. for each, invert one randomly selected bit in its representation.
 4. Update: $P \rightarrow P_S$
 5. Evaluate: for each h in P , compute $Fitness(h)$.
- Return the hypothesis from P that has the highest fitness

Note that the *Crossover* and *Mutate* steps should be implemented in such a way that prohibit that invalid strings are generated. Note also that a genetic algorithm typically uses a population of bitstrings as its input, so the notion of a *string* here is really a bitstring. E.g. to optimize a program with 10 objects for an architecture with 4 banks, we'd use bitstrings of length 20 (2 bits per object). If the number of banks is not a power of 2, there's an additional case to check for when trying to avoid generating invalid bitstrings.

3.2.2 Speeding up the Algorithm

There are several ways which may decrease the execution time of the search:

- The way the *Cost* function is implemented (i.e. as a transformation or equivalently as a finite state machine) is important because of the possible simplifications that can be made. The problem is that calculating the τ function (and hence calculating the penalty of all memory conflicts) will take too long, even if we use the simplification mentioned above. Therefore, the mechanics of τ itself need to be simplified, i.e. certain cases that give rise to memory conflicts will have to be neglected. Since τ is a finite state machine, this will result in getting rid of certain states.

- Calculating τ is a costly operation since it involves going over all instructions of T_{ep} . Optimizing the memory layout will involve calculating τ a (potentially very large) number of times. Therefore, T_{ep} is divided into segments and τ is approximated by calculating τ_i for each segment S_i in T_{ep} . Accordingly, $\tau = \sum_{i=1}^{N_S} \tau_i$ is used, where N_S is the number of segments in T_{ep} . The memory layout of these segments is then optimized individually. This way, those conflicts that may exist between segment borders are missed, but this is a loss that can be minimized by picking appropriate segments. Instead of just fixing the size of a segment, basic blocks are chosen as segments. This has the benefit towards the efficient calculation of τ that a lot of segments (basic blocks) in T_{ep} are the same, e.g. when a loop is executed. Indeed, calculating τ for the loop body once is enough if the body never changes. However, care must be taken here since there is a case where this can go wrong. Consider the following extract from a loop body:

```

...
mmem (r5)++,r6
...

```

The first time, register r5 will point to an object o_1 and in the consecutive execution of the loop body it will point to an object o_2 . Now what if the DSP architecture for which the program (or more precisely: the data allocation) must be optimized has a continuous address space where the address of o_1 is in memory bank b_i and the address of o_2 (which is the *next* address) is in b_j ($i \neq j$)?

3.3 A Fast and Practical Solution

3.3.1 Reconsidering τ : the Similarities with Register Allocation

A good starting point for finding a better way to tackle the problem at hand is to reconsider what information running τ over a given program can supply. While evaluating τ , a datastructure that holds information on the conflicts between various objects can easily be updated. This information can, if needed, be very detailed: for each object, it is not only possible to collect the number of conflicts with each other object but also the nature of those conflicts. The first question to ask then, is: given the task to perform, what part of the information is useful and what part is not?

- **the number of conflicts** (or even more precise: the number of lost cycles due to the presence of those conflicts) between 2 objects is vital as it leads to the natural heuristic that the more they interfere¹¹, the more important it is for them to be allocated in a way that either the number of these conflicts or the negative effects of those conflicts are minimized.
- **the number of uses of each object** is also relevant: the more an object is accessed from memory, the more important it is that it resides in a bank where it can be accessed with the least possible loss of cycles. Frequently used objects must be assigned to fast banks.
- **the size of the objects**: although the optimal choice for an object o may be to reside in bank b_k , if $Size(o) > Size(b_k)$ it's just not possible without changing (part of) the code (e.g. split an array into several parts). Also, bigger objects may have to be assigned first since

¹¹Analogous to what is known in register allocation ([Chaitin, 81], [Chaitin, 82], [Briggs, 92]), we say that two objects o_i and o_j *interfere* if there exists at least one conflict between them during the execution of the program.

doing this later on may severely limit the placement options: in figure 1 (a), object o is too big to fit into B_2 ; in (b), o_5 doesn't fit in either bank, but if we had e.g. assigned o_4 to B_1 instead of B_2 , o_5 could have fit into B_2 . In figure 1 (b) seems to indicate that the size of an object must play a certain role during the decision process of the order in which objects are assigned to a bank: larger objects must be assigned early enough to guarantee that they fit in somewhere. However, under the assumption that there always is a large enough (slow) memory the object could fit in, it is obvious that we don't need to take o_{size} into account: if an object is not used very often and it doesn't cause a lot of conflicts, it doesn't matter if we have to put it into a slow bank due to its size. Putting it in a smaller but faster bank early will probably cost us too much later on, because a lot of often used or highly conflicting objects that will then have to be put into the larger but slower bank.

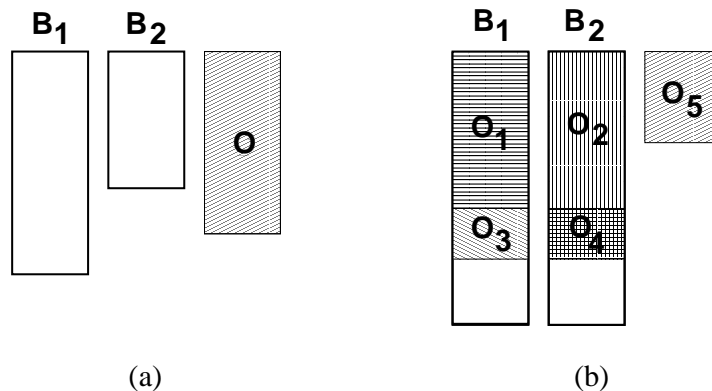


Figure 1: Placement problems with object size.

- **the nature of the conflict** (e.g. *program wait*¹², *program fetch incomplete*¹³, ...) is unimportant. Basically, what happens is reading or writing (parts of) objects. If the assumption is made that every memory bank has both read and write access and reading a word from memory and writing a word to it are equally fast, conflict natures can safely be ignored. This assumption is valid because the memory architecture of a DSP processor must be general enough to be able to do a lot of different tasks.
- **the structure of the pipeline** is hidden by applying τ which gives an overview of the conflicts that particular pipeline design would cause. Therefore, we don't have to take it into account anymore.
- **the memory architecture itself**: this is obviously a major factor.

The problem considered in this report is similar to the *storage bandwidth optimisation* problem as described in [Catthoor, 98], but their input is different (C source code instead of an execution trace) and the constraints they have to take into account are different also: while we try to put data into a given memory architecture, they try to derive an optimal memory architecture to be able to run the code within a given cycle budget.

Given the facts cfr. supra, the datastructure can be kept very simple. We will build an *interference graph*¹⁴ $G = (N, E)$ where each node n_i of the set of nodes N represents an object

¹²An instruction fetch must be delayed because there are too many accesses to the same memory that the instruction must be fetched from or a multicycle data access is needed.

¹³Fetching the instruction takes more than 1 cycle because of memory waitstates.

¹⁴Cfr. supra, we again adapt the terminology widely used in the domain of register allocation.

of the program¹⁵ and each edge e_j of the set of edges E is a labeled, undirected arc (between 2 or more edges¹⁶) where the label is the total number of possible conflicts between the objects represented by the nodes that are connected by the edge. Note that a node can have an edge to itself, i.e. *self-referencing edges may exist*. In figure 3, object C has a self-referencing edge indicating that during some cycle C is accessed twice. The reason we add hyper edges to the graph is that it is important to see the actual conflicts in the graph: later on, we will remove edges from the graph one by one, each edge representing a conflict. As some conflicts have a bigger influence on performance than others, it will be important to resolve them in order of importance (i.e. give priority to the conflict that implies the heaviest penalty). As a small example, consider the two graphs from figure 2 where (a) is the version without hyper edges and (b) is the version with hyper edges included. From (b) we learn that there are actually only two conflicts, while looking at (a) leads us into the false conclusion that there are 6. While the additional benefits of hyper edges may not be immediately obvious, they can lead to interesting additional benefits. Consider the example in figure 3 where in (b) part of the conflicts from (a) are represented by a hyper edge of degree 3¹⁷ so at a certain moment during the program execution, we need one access to B and two accesses to C during the same cycle. Imagine that in our assignment algorithm we have the following situation: when resolving graph (a), first all conflicts corresponding to edges of degree 2 are resolved and afterwards the self-referencing conflict is looked into. We might already have decided to put C into a slower bank (e.g. only 1 access per cycle) and due to other assignments there may no longer be a place for C in a faster one. In (B) this problem is avoided: if the hyper edge conflicts are resolved first, C is automatically placed in a faster bank. This illustrates the extra knowledge gained by introducing hyper edges: the relations between certain conflicts are now visible in the graph while without hyper edges they are not.

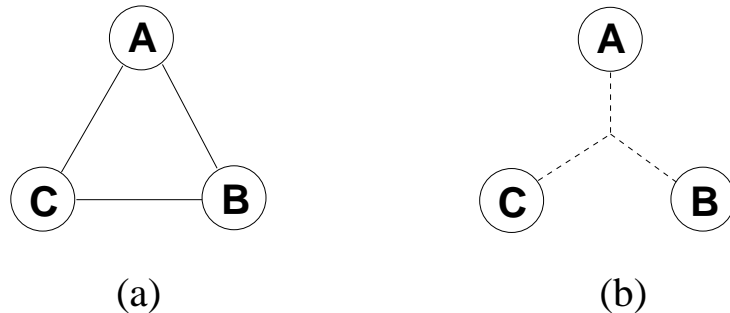


Figure 2: Hyper edges introduce additional (i.e. more precise) knowledge.

The fact that the representation of the problem is so similar to what was introduced for register allocation was already hinted at by the proof of the NP-completeness where we mapped our problem onto the k -colorability problem. The equivalence is not trivial though, since the concept of *spilling*¹⁸ is a local phenomenon in register allocation (with a cost that is known) while for our memory layout problem it has a more global nature. Spilling a variable that is only known inside a loop will result

¹⁵In the rest of the text, the concepts *node* and *object* will be used interchangeably

¹⁶We call edges using connecting more than 2 nodes *hyper edges* and will represent them by dotted lines.

¹⁷We define the *degree of a hyper edge* to be the number of nodes connected by it. The degree of a self-referencing edge is 2.

¹⁸In register allocation, a variable is spilled if no suitable register can be found for it. For each variable, spill code is inserted into the program. Basically, each *use* of the variable is preceded by a *load* and each *def* is followed by a *store*. Here, spilling an object can be interpreted as having to put that object into a bank that is not optimal for it in the sense that it is still involved in some conflicts.

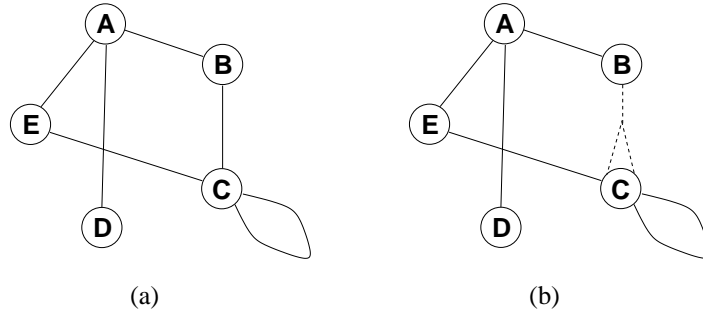


Figure 3: Hyper edges introduce additional benefits.

in some `load` and `store` instructions locally to that loop. However, if we have to put the same variable into a different memory bank, this may affect the entire program (spilling it to that bank may result in a lot of extra conflicts). Also note that the nature of the problem we're solving bears more similarity to *register assignment* than register allocation¹⁹.

3.3.2 The Interference Graph Part 1: Building it Up

The problem with the above is that an interference graph for a particular allocation was considered. What is really needed is an interference graph for all possible conflicts, i.e. for the worst allocation possible. Given such a graph, the objective is then to try to color the nodes of it where coloring a node with a certain color is equivalent with assigning the object related to the node to a certain memory bank. If τ is slightly modified to just go over T_{ep} and report any possible memory conflict between two objects to the interference graph (which is then updated accordingly), the worst case scenario is effectively modeled (just place every object in the same - and slowest - memory bank and all possible conflicts will be actual conflicts). Since the number of *uses* of a certain object is also important, a label is attached to each node which represents exactly that. By using the information this graph provides, it should be possible to build up a good allocation. For that to work, some good heuristics are needed to indicate which objects will be allocated first. The importance of allocating a node before other nodes is that this allocation (possibly) puts extra constraints on the subsequent allocations of those remaining nodes (e.g. they might not fit into a certain bank anymore because of the size of the object that just got put there).

3.3.3 The Interference Graph Part 2: Breaking it Down

When considering breaking down the interference graph, the vital question is: will the focus be on the nodes or on the edges? Removing a node from the graph corresponds to assigning that node to a certain memory bank. Removing an edge, however, corresponds to resolving a number of conflicts between various nodes: if all the nodes that are connected by the edge under consideration are assigned to a memory bank, we have actually figured out whether a given number of conflicts can or can not be resolved. Therefore, removing edges seems the logical thing to do: if the goal is to assign a certain object to a memory bank, it's highly desirable that this be done with respect to the assignment of other, interfering objects. With this issue resolved, the next problem to address is deciding in what order the edges should be removed. Removing an edge involves assigning the nodes it connects to a memory so some nodes will be assigned earlier than others, which may lead

¹⁹While *register allocation* deals with deciding which variables will reside in a register and which will not, the *register assignment* process determines in which physical registers each allocated variable will reside.

T_2	
f	number of lost cycles
$f_1(b_1, b_1)$	0
$f_2(b_1, b_2)$	0
$f_3(b_1, b_3)$	2
$f_4(b_2, b_2)$	0
$f_5(b_2, b_3)$	2
$f_6(b_3, b_3)$	3

Table 1: Table for conflicts of degree 2.

to a different result compared to what it would have been if the order was different. In order to get as close as possible to the optimal solution, as mentioned above, some good heuristic(s) are needed to determine which conflicts to resolve early on and which to leave until the end. We will address this problem by taking a look at what we gain by resolving every conflict in the graph. The gain is trivially expressed in terms of cpu cycles. If we have an interference graph where the degree of the edges is in the range $2..n$, we build $n - 1$ tables T_2, \dots, T_n where T_i has $\binom{\#Banks + i - 1}{i} = \frac{(\#Banks + i - 1)!}{i! (\#Banks - 1)!}$ entries²⁰. Each entry in T_i is a function

$$f(b_1, \dots, b_i) : Banks \times Banks \times \dots \times Banks \rightarrow N$$

where $b_j \in Banks$ ($j \in [1..i]$) and $f(b_1, \dots, b_i)$ is the number of cycles lost 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, \dots and one object is in b_i . If e.g. $Banks = \{b_1, b_2, b_3\}$ where

- $ca(b_1) = 1$
- $ac(b_1) = 2$
- $ca(b_2) = 1$
- $ac(b_2) = 2$
- $ca(b_3) = 2$
- $ac(b_3) = 1$
- b_1 and b_2 can be accessed in parallel

then T_2 ($\binom{4}{3} = 6$ entries) and T_3 ($\binom{5}{3} = 10$ entries) are defined as shown in table 1 and table 2 respectively:

For each edge in the conflict graph, we consult table T_i with I the degree of the edge. If we do this for all edges, we can easily find which conflict resolution will give us the most benefit and in which memory banks the objects involved in the conflict need to be placed for an optimal result. For an edge e connecting n nodes N_1, \dots, N_n where nrc conflicts occur between all nodes at once and

²⁰ i -combinations with repetition of $\#Banks$

T_3	
f	<i>number of lost cycles</i>
$f_1(b_1, b_1, b_1)$	1
$f_2(b_1, b_1, b_2)$	1
$f_3(b_1, b_1, b_3)$	2
$f_4(b_1, b_2, b_2)$	1
$f_5(b_1, b_2, b_3)$	2
$f_6(b_1, b_3, b_3)$	4
$f_7(b_2, b_2, b_2)$	1
$f_8(b_2, b_2, b_3)$	2
$f_9(b_2, b_3, b_3)$	4
$f_{10}(b_3, b_3, b_3)$	5

Table 2: Table for conflicts of degree 3.

the total number of other uses of node N_i (i.e. uses that are not part of the conflict e corresponds to) is $uses(N_i)$, a good initial heuristic might be:

$$BEST_e = f_j(b'_1, \dots, b'_n) \times nrc + \sum_{i=1}^n uses(N_i) \times ca(b'_i)$$

where

$$f_j(b'_1, \dots, b'_n) = \min_j f_j \in T_n$$

What this formula expresses is actually just the summation of two important factors in our decision making: the total number of cycles that are lost due to assigning the objects N_1, \dots, N_n to banks b'_1, \dots, b'_n respectively when the nrc conflicts of edge e occur, plus the total number of cpu cycles for all other accesses of the objects (whether in another conflict or isolated). The latter factor is not calculated in terms of *lost* cycles because this would all too often be 0, which would render that part of the cost function useless.

Similarly, we might calculate the loss of cpu cycles for this situation when all objects would be assigned to the largest, slowest memory b_{slow} :

$$WORST_e = f_j(b_{slow}, \dots, b_{slow}) \times nrc + \sum_{i=1}^n uses(N_i) \times ca(b_{slow})$$

which is an upper limit to how bad we can do. As for breaking down the conflict graph, the criterion can be either one of these (or, for that matter, any other criterion that expresses something along the same lines):

1. resolve the conflict with the biggest possible benefit first
2. resolve the conflict with the biggest possible worst case first

These are not necessarily equivalent, since large, often used objects might not fit into the fastest memory bank.

If we now iterate over all edges e and calculate

$$GAIN_e = WORST_e - BEST_e$$

for each of them, we have an indication that, if for e_i $GAIN_{e_i}$ is maximal, than e_i is the conflict to resolve first. The idea is that in that case, we have the smallest loss of cpu cycles²¹. Exactly how this all works out then, is illustrated in an example below.

3.3.4 An Example

Given an architecture with only a 2 stage pipeline, 1 three operand²² instruction `instr src1 src2 dst` and 3 memory banks ($Banks = \{b_1, b_2, b_3\}$) with restrictions as described cfr. supra and in table 1 and table 2. Further details of interest are:

- in stage 1 of the pipeline the objects (if any) in `src1` and `src2` are accessed
- in stage 2 of the pipeline, the object (if any) in `dst` is accessed
- one memory cell (of any bank) can hold 1 instruction
- $Size(b_1) = \#b_1 = 22$
- $Size(b_2) = \#b_2 = 22$
- $Size(b_3) = \#b_3 = 1000$

consider the following program with 8 objects ($Objects = \{A, B, C, D, E, F, G, H\}$):

```
instr #A, 2, 3
instr 2, #A, 5
instr 5, 5, #A
instr -5, 0, 0
instr 2, #B, 3
instr #B, #C, #D
instr #D, #C, #C
instr #D, 3, #B
instr #D, #E, 0
instr #B, 1, 7
instr 2, #C, #E
instr #F, #C, #E
instr #C, #C, #C
instr 2, #C, #F
instr #B, 3, #B
```

²¹Of course, we will need some sort of tie-breaker mechanism since the same value for GAIN may occur more than once.

²²An operand can either be an immediate or an address. If we write $\#O$, we mean that the operand is an address that refers to a memory cell where a part of object O is stored.

```

instr #G, #C, #E
instr #G, -1, #E
instr #E, #H, 5
instr 1, 4, 0
instr #E, #G, #C
instr 0, 0, #B

```

where: $Size(A) = 1$, $Size(B) = 7$, $Size(C) = 15$, $Size(D) = 2$, $Size(E) = 2$, $Size(F) = 6$, $Size(G) = 16$ and $Size(H) = 5$. This results in the *interference graph* of figure 5 if nhyper edges are allowed and the one of figure 4 if no hyper edges are used. The number inside each node represents the number of uses of the object, the label on each edge is the number of conflicts between those objects.

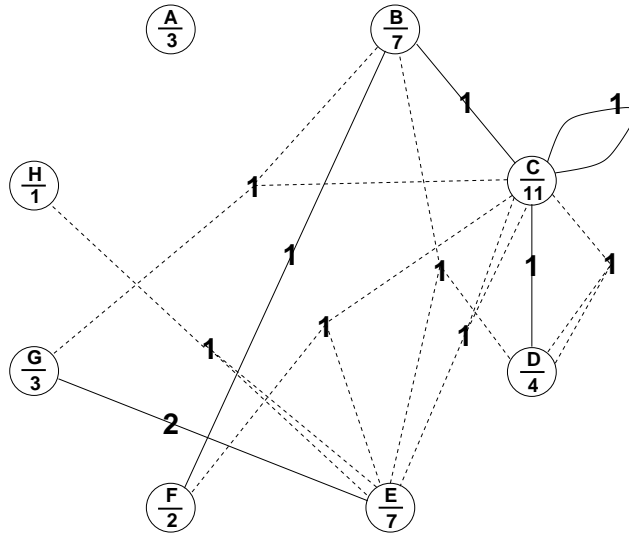


Figure 4: Conflict graph without hyper edges for the example from 3.3.4

If we use figure 5 as our conflict graph, we obtain the information as summarized in table 3 where *nodes* lists the nodes involved in a particular conflict (conflicts where a certain object is involved n times are listed that many times), *conflicts* is the number of occurrences of this conflict, *total uses* is the total number of accesses to the objects involved in the conflict (total, i.e. for the entire program!) and *uses \notin conflict* is the total number of times each object of the conflict is used outside of the conflict described in that row of the table. Table 4 then shows the value of $GAIN_e$ for each edge e ²³.

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

1. Remove edge (C, C) from the graph and assign C to b_1 .
2. Remove edge (B, C, G) from the graph and assign B to b_1 and G to b_2 . b_1 is now completely filled and b_2 has 6 words of free space left. Remove edge (B, C) because all its nodes have been allocated to a bank.
3. Remove edge (C, E, F) from the graph and assign E to b_2 and F to b_3 . We take E over F to be assigned to b_2 because E is used more often (7 uses for E compared to 2 uses for F). Remove edges (C, C, E) , (G, E) and (B, F) .

²³Entries in the table are sorted in descending order according to their $GAIN_e$ value.

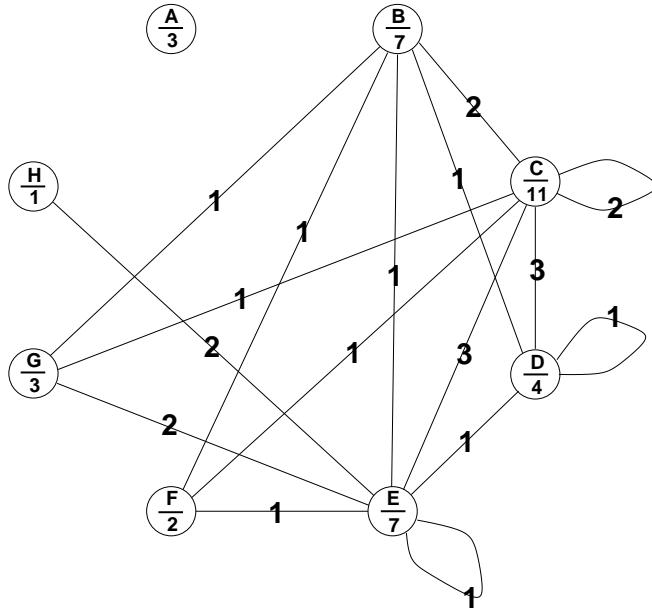


Figure 5: Conflict graph with hyper edges for the example from 3.3.4

<i>nodes</i>	<i>conflicts</i>	<i>total uses</i>	<i>uses \notin conflict</i>
C, C	1	$11 + 11 = 22$	$10 + 10 = 20$
B, C	1	$7 + 11 = 18$	$6 + 10 = 16$
G, E	2	$3 + 7 = 10$	$1 + 5 = 6$
B, F	1	$7 + 2 = 9$	$6 + 1 = 7$
C, D	1	$11 + 4 = 15$	$10 + 3 = 13$
B, D, E	1	$7 + 4 + 7 = 18$	$6 + 3 + 6 = 15$
B, C, G	1	$7 + 11 + 3 = 21$	$6 + 10 + 2 = 18$
C, C, E	1	$11 + 7 = 18$	$9 + 6 = 16$
C, E, F	1	$11 + 7 + 2 = 20$	$10 + 6 + 1 = 17$
C, D, D	1	$11 + 4 = 15$	$10 + 2 = 12$
E, E, H	1	$7 + 1 = 8$	$5 + 0 = 5$

Table 3: Relevant conflict information.

<i>nodes</i>	<i>conflicts</i>	$WORST_e$	$BEST_e$	$GAIN_e$
C, C	1	$1 \times 3 + 20 \times 2 = 43$	$1 \times 0 + 20 \times 1 = 20$	$43 - 20 = 23$
B, C, G	1	$1 \times 5 + 18 \times 2 = 41$	$1 \times 1 + 18 \times 1 = 19$	$41 - 19 = 22$
C, E, F	1	$1 \times 5 + 17 \times 2 = 39$	$1 \times 1 + 17 \times 1 = 18$	$39 - 18 = 21$
C, C, E	1	$1 \times 5 + 16 \times 2 = 37$	$1 \times 1 + 16 \times 1 = 17$	$37 - 17 = 20$
B, C	1	$1 \times 3 + 16 \times 2 = 35$	$1 \times 0 + 16 \times 1 = 16$	$35 - 16 = 19$
B, D, E	1	$1 \times 5 + 15 \times 2 = 35$	$1 \times 1 + 15 \times 1 = 16$	$35 - 16 = 19$
C, D	1	$1 \times 3 + 13 \times 2 = 29$	$1 \times 0 + 13 \times 1 = 13$	$29 - 13 = 16$
C, D, D	1	$1 \times 5 + 12 \times 2 = 29$	$1 \times 1 + 12 \times 1 = 13$	$29 - 13 = 16$
G, E	2	$2 \times 3 + 6 \times 2 = 18$	$2 \times 0 + 6 \times 1 = 6$	$18 - 6 = 12$
B, F	1	$1 \times 3 + 7 \times 2 = 17$	$1 \times 0 + 7 \times 1 = 7$	$17 - 7 = 10$
E, E, H	1	$1 \times 5 + 5 \times 2 = 15$	$1 \times 1 + 5 \times 1 = 6$	$15 - 6 = 9$

Table 4: Heuristically deciding the order for conflict resolution.

4. Remove edge (B, D, E) from the graph and assign D to b_2 , leaving 4 words of free space in that bank. Remove edges (C, D) and (C, D, D) .
5. Remove edge (E, E, H) from the graph and assign H to b_3 .
6. Assign A to b_2 .

The final allocation is represented in figure 6.

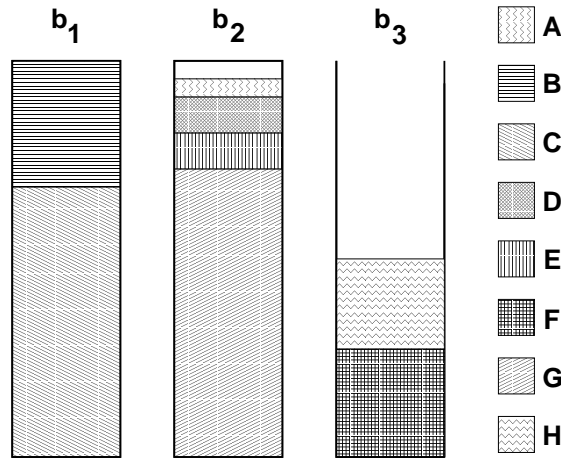


Figure 6: Conflict graph without hyper edges for the example from 3.3.4

4 Some Notes of Interest

Relation to link-time optimization Our optimization technique is not suitable for use by the compiler as it needs the exact machine instructions as they are executed by the processor. However, this technique could be embedded into the linker. After the introduction of the idea in [Wall, 86], several link-time optimizers (and related to that, whole program optimizers) have been

build, e.g. OM ([Srivastava, 93], [Srivastava, 94]), Alto ([Muth, 98], [Muth, 99]), Spike ([Cohn, 97], [Goodwin, 97]), MLD ([Fernandez, 96]) and Vortex ([Chambers, 96]).

Other search strategies and/or criteria Instead of using a genetic algorithm to optimize the τ function, could other search strategies have been used? An excellent overview of algorithms to optimize a function is given in chapter 10 of [Numerical Recipes, 92]: *Simplex, Powell, Gradient Descent, ...* The main reason for avoiding these methods is that τ is not a continuous function so there is not really the notion of a *sense of direction* (or *gradient*) here: allocating just one object differently may result in a totally different evaluation value. Another very interesting idea could be to use the notions of *entropy, joint entropy* and *mutual information* ([Reza, 94]). Entropy is a measure of randomness. The more random a variable is, the more entropy it will have. *joint entropy* is a statistics that summarizes the degree of dependence of X and an other random variable Y. Mutual information then is a measure of the reduction of the entropy of a variable Y given another variable X. This might work to solve our problem in the following way: if we place one object o in a certain bank b , we can calculate the mutual information and/or entropy between o and the other objects o_i . The probability distributions will have to be estimated according to notions of *object importance* and the information we have in the *BankChars* and *BusChars* set. This may be a non-trivial task, but if we could maximize the mutual information or maximize the entropy between all objects, we would have a solution to our problem. If the entropy is maximal, the objects interfere as little as possible, which corresponds to a minimization of conflicts, just the thing we need.

Another point of interest is that optimization can be done with respect to a lot of different criteria. In the DSP context, power use is one of them. Fact is that a minimization of the power use of the memory is highly desirable for e.g. DSP processors in GSM mobile phones. Using the same τ but a different *Cost* function should enable us to optimize for this criterion using the same framework.

Topics for (immediate) future research Other ideas that need to be looked into are the concept of *stride*, the connection between the *next_io* function and the fact that the trace is *manifest*, as well as the concept of *lifetime of a variable*. The latter is closely related to scheduling. Instruction rescheduling could be an intermediate step between the current work and actually changing the instructions themselves and could perhaps remove some conflicts, resulting in a simpler conflict graph and possibly a better allocation.

Also, we've been working with T_{ep} for reasons of correctness, but working with just T would be a good way of decreasing the execution time needed to evaluate τ . The reason this is valid, is that most processors these days have near perfect branch prediction. Of course, putting T_{ep} aside in favor of T is only important for the (near) optimal solution search methods. If only one execution of τ is needed, it doesn't matter that it takes a bit longer to complete.

Finally, some thought must be given to duplicating (or cloning) objects in memory. A possible case where this can be interesting is shown in figure 7. *Banks* = $\{b_1, b_2\}$ where the banks can be accessed in parallel during one cpu cycle and $ac(b_1) = ac(b_2) = 1$. If there are conflicts between objects A and C but also between B and C , we can solve this by duplicating C and putting it in both banks. Then, we can modify the address of the accesses to C in the code to make sure the benefits from parallel access are maximized.

As for now, we do not take the size of an object into account when deciding when to resolve certain conflicts before others. There are of course situation where this can lead to very bad results. For example, take the conflict graph from figure 8: the conflict between objects A and B is resolved

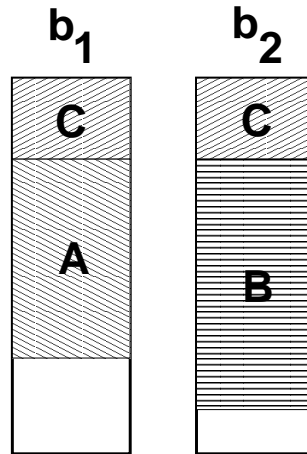


Figure 7: Duplicating objects to maximize parallelism.

first. This causes all other objects to be moved to the large but slow memory bank, while all of them would have easily fitted into the smaller banks. Cases like this must definitely be dealt with and will require further tuning of our criterion. This tuning will probably be application-type dependent, as the data requirements for dsp applications are typically that one large object (the input signal) needs to be accessed during almost the entire execution of the program, while for multimedia applications there will be a lot of objects with a short lifetime.

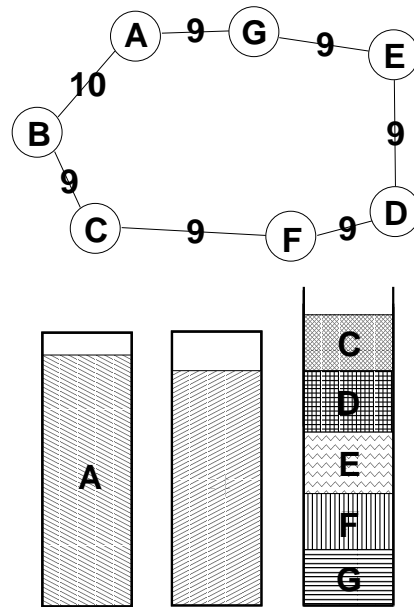


Figure 8: The importance of object sizes.

Finally, some sort of backtrack mechanism could be interesting, since at the moment there is no way of undoing a decision made earlier during the process if later on it becomes obvious that that local allocation decision has very negative consequences for the quality of the overall allocation. Perhaps we should also allow a few extra evaluations of τ at certain points during the algorithm if very important decisions have to be made at those points: the benefits of making a good decision

there may outweigh the additional cost of these evaluations.

Acknowledgements

This work was sponsored by the Fund for Scientific Research - Flanders under grant 3G001998. Also, personal correspondence with Saumya Debray of the University of Arizona was greatly appreciated.

References

- [Briggs, 92] P. Briggs. Register Allocation via Graph Coloring. Phd. thesis, Rice Univ., Houston, TX, Apr. 1992.
- [Cattoor, 98] F. Cattoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle. *Custom Memory Management Methodology* Kluwer Academic Publishers, 1998.
- [Chaitin, 81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, P. Markenstein. Register Allocation Via Coloring, *Computer Languages*, Vol. 6, No. 1, 1981, pp.47-57.
- [Chaitin, 82] G. Chaitin. Register Allocation and Spilling via Graph Coloring. *Proc. of the SIGPLAN '82 Symp. on Compiler Constr.*, Boston, MA, published as *SIGPLAN Notices*, Vol. 17, No. 6, June 1982.
- [Chambers, 96] G. Chambers, J. Dean, and D. Grove. Whole-program optimization of object-oriented languages. Technical Report 96-06-02, Department of Computer Science and Engineering, University of Washington, June 1996.
- [Cohn, 97] R. Cohn, D. Goodwin, P.G. Lowney, and N. Rubin. Spike: An optimizer for alpha/NT executables. In *USENIX Windows NT Workshop*, August 1997.
- [Fernandez, 96] M.F. Fernandez. *A Retargettable, optimizing linker*. PhD thesis, Princeton University, January 1996.
- [Goodwin, 97] D.W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 122-133, June 1997.
- [Kozen, 92] D. C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.
- [Mitchell, 97] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [Muth, 98] R. Muth., S. Debray, S. Watterson, and K. De Bosschere. *alto*: A link-time optimizer for the dec alpha. Technical Report 98-14, The University of Arizona, 1998.
- [Muth, 99] R. Muth. *ALTO: A Platform for Object Code Modification*. PhD thesis, The University of Arizona, August 1999.
- [Numerical Recipes, 92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery *Numerical Recipes in C, Second Edition* Cambridge University Press, 1992.
- [Reza, 94] Reza, F.M. *An Introduction to Information Theory* Dover, New York, 1994.

- [Srivastava, 93] A. Srivastava and D.W. Wall. A practical system for intermodule code optimization at link-time. In *Journal of Programming Languages*, pages 1-18, March 1993. Also available as WRL Research Report 92/6.
- [Srivastava, 94] A. Srivastava and D.W. 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. Also available as WRL Research Report 94/1.
- [Texas Instruments, 99] *TMS320C4x General-Purpose Applications User's Guide* Texas Instruments, pp. 5-1/.../5-6, 1999.
- [Texas Instruments, 97] *TMS320C3x/C4x Optimizing C Compiler User's Guide* Texas Instruments, pp. 4-2, 1997.
- [Wall, 86] D.W. Wall. Global register allocation at link time. In *Proceedings of the ACM SIGPLAN '86 Conference on Compiler Construction*, pages 264-275, 1986. Also available as WRL Research Report 86/3.