Run-Time Memory Optimization for DDMB Architecture Through a CCB Algorithm

Jeonghun Cho[1] and Yunheung Paek[2]

[1] School of EECS, Kyungpook National University, 1370 Sangyuk-dong, Buk-gu, Daegu, Korea
jcho@ee.knu.ac.kr

[2] School of EECS, Seoul National University, San 56-1, Sillim-dong, Gwanak-gu, Seoul, Korea
ypaek@ee.snu.ac.kr

Abstract. Most vendors of digital signal processors (DSPs) support a Harvard architecture, which has two or more memory buses, one for program and one or more for data and allow the processor to access multiple words of data from memory in a single instruction cycle. We already addressed how to efficiently assign data to multi-memory banks in our previous work. This paper reports on our recent attempt to optimize run-time memory. The run-time environment for dual data memory banks (DDMBs) requires two run-time stacks to control activation records located in two memory banks corresponding to calling procedures. However, activation records of two memory banks for a procedure are able to have different size. As a consequence, dual run-time stacks can be unbalanced whenever a procedure is called. This unbalance between two memory banks causes that usage of one memory bank can exceed the extent of on-chip memory area although there is free area in the other memory bank. We attempt balancing dual run-time stacks to enhance efficiently utilization of on-chip memory in this paper. The experimental results have revealed that although our call chain balancing (CCB) algorithm is relatively quite simple, it still can utilize run-time memories efficiently; thus enabling our compiler to run extremely fast, yet minimizing the usage of run-time memory in the target code.

Keywords: Run-time environment, DSP, dual data memory banks, compiler, and on-chip memory.

1 Introduction

To reduce the speed gap between a processor and memory subsystems, DSP architectures commonly include on-chip memory. This memory is configured as a dual-bank for a Harvard architecture that allows simultaneous program and data memory access [6]. Recently, several DSP products, such as Analog Device ADSP2100, DSP Group PineDSPCore, Motorola DSP56000 and NEC uPD77016, have even enhanced the original design of the Harvard architecture by providing one additional data bank. This enhancement is targeted to effectively support common DSP functions such as a FIR filter.

$$c(n) = \sum_{i=0}^{N-1} (a(i) \times b(n-i))$$

Note that this on-chip memory is small and therefore, the efficient use of memory turns out to be the major challenge in DSP software development process. To partially address this issue, we developed a compiler strategy that assigns variables to multiple data memory banks such that overall system performance can be improved; Refer [2] for a detailed introduction. However, recent compiler validation efforts lead us to observe that even with our previous compiler techniques; The lack of an efficient compiler strategy, which can manage the *runtime environment* for variables in dual data memory banks, significantly hurts the overall DSP system performance.

For this particular problem, this paper presents a scheme to efficiently manage local variables stored in each *activation record* (AR) in the dual data memory banks (DDMBs) of a commercial DSP. Our experiment reveals that this scheme greatly helps the compiler to balance the total amount of memory space for ARs stored in the DDMBs, thus better utilizing the on-chip SRAM of the DSP, which in turn helps us to maximize the performance on our target DDMB architecture. For this work, we first apply our dual memory bank assignment technique [2] to assign local variables in a procedure into two ARs for the procedure, each of which must be stored in either memory bank X or Y, respectively. Then, as will be shown later, our scheme attempts to store the ARs such that both the banks are equally utilized and the amount of wasted memory space is minimized.

In Section 2, we provide a brief overview of previous approaches for the MBA problem, and discuss their schemes to store ARs in multiple memory banks. To this end, we also discuss several strategies for runtime memory management. In Sections 3 and 4, we define the problem of optimally storing ARs into the DDMB architecture, and show our approach to address this problem. In Section 5, we present the experimental results with a set of DSP benchmarks on a commercial DSP with DDMB architecture, and compare our performance with others.

2 Previous Work

In the MBA problem, we are required to divide all local and arguments variables referenced in each procedure into two groups, and to allocate each group to either one of the data memory banks. One approach to address the MBA problem was done by [5]. In their approach, a data dependency graph (DDG) is first constructed for each assembly code function. For a given DDG, the interference graph is constructed in such a way that potential parallelism is reflected by graph edges. After a step for reducing the interference graph size, they applied an Integer Linear Programming (ILP) to partitioning for the interference graph. The SPAM project was also conducted to solve this assignment problem by researchers at Princeton and MIT [8]. In their work, they presented experimental results showing that the SPAM compiler can generate highly optimized code for a commercial DSP in most cases. However, their approach as well as the above ILP-based one has a drawback that the compilation time may increase substantially for large applications. To reduce compilation time, we proposed in [2] a fast approach based on a *maximum spanning tree* algorithm to solve multiple memory bank assignment in polynomial time.

To utilize efficiently assigned memory banks, well-designed run-time environment is indispensable. The representative run-time environment for the C programming language is the stack-based environment, whose essential structure does not depend on the specific details of the target machine. However, if the target machine such as Motorola DSP56300 has two data memory banks, it would be more desirable to assume two run-time stacks in the environment. Unfortunately, simultaneously managing multiple run-time stacks along with multiple frame pointers and stack pointers give rise to various complex optimization issues, as compared to conventional single run-time stacks. To simplify these issues, the SPAM compiler [8] uses a fully static run-time environment (see Fig. 1 (a)), instead of the stack-based environment. In the static environment, all ARs must be stored at some fixed position in the memory before the program actually runs. Therefore, it can be easily implemented and there is no need to maintain multiple pointers to keep track of execution path at run time. However, one critical disadvantage is that a large waste of memory can occur because all ARs must be always allocated at run time.

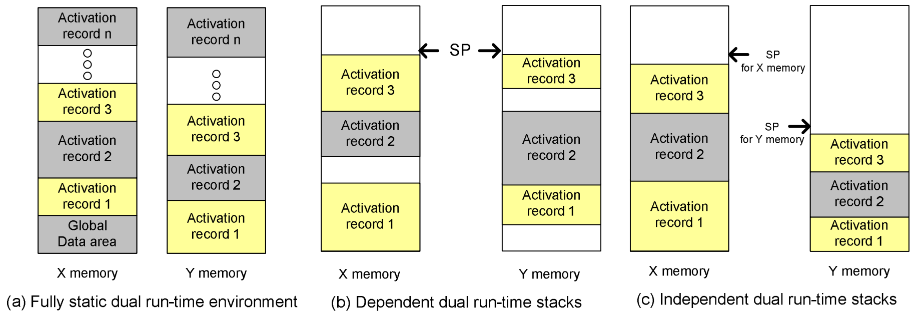


Fig. 1. Run-time Environments for DDMB

In our earlier work [2], we attempted to overcome these disadvantages by maintaining dual run-time stacks, as illustrated in see Fig. 1 (b). One clear advantage of this stack implementation is that we only require only one stack pointer as well as one frame pointer just like in the case of a single stack implementation. Also, the scheme to maintain dual stacks is almost as simple as that to maintain a single stack. However, as shown in Fig. 1 (b), there is still some unnecessary waste in memory in order to synchronize both the stacks. To alleviate this drawback, we in this work propose a scheme where the two stacks are managed independently as shown in Fig. 1 (c). To strike the difference between these stack implementations, we call the earlier one as the *dependent stack* implementation while we call the later one as the *independent stack* implementation. Although the AR assignment to independent stacks normally results in better memory utilization, a sophisticated scheme to balance the sizes of the two stacks is mandatory to enjoy the advantage. Otherwise, we may suffer a degradation of performance since the larger stack in either side of the banks grows out of the boundary of on-chip SRAM, thereby causing frequent accesses to off-chip memory. Fortunately, the size of an AR can be determined at compile time since the compiler decides its fields and their sizes. Using this formation, we were able to pre-compute the overall requirement for memory use at compile time, enabling us to carefully balance the dual stacks.

3 Motivation and Approach

In this section, we discuss how run-time memory can be optimized in DDMB and what the benefits are, especially for run-time memory. First, we describe our motivation to optimize usage of run-time memory, and then we provide our approach briefly to enhance a performance.

3.1 Our Motivation

Our run-time environment for DDMB uses three memory banks: program memory, X memory, and Y memory. Generated code from a compiler is located in the program memory, and all static and dynamic data are located in X or Y memory. Global and static variables are assigned in the global/static area in X and Y memory, and local variables and information to manage activations of procedures are assigned in the stack area. Because our compiler does not support variable allocated dynamically (ex. *malloc* function in C programming language), the heap area are excluded in our run-time environment. We start with the definition of balancing for dual run-time stacks.

Definition 1. *Let* $A = \{\{A_{i0}, A_{j0}\}, \{A_{i1}, A_{j1}\}, \ldots, \{A_{in-1}, A_{jn-1}\}\}$ *be a set of n activation records in the run-time stack at any instant where* A_{ik} *and* A_{jk} *mean the activation records for two memories of the function k* $(0 \leq k \leq n-1)$*. And let* $AX = \{A_{X0}, A_{X1}, \ldots, A_{Xn-1}\}$ *and* $AY = \{A_{Y0}, A_{Y1}, \ldots, A_{Yn-1}\}$ *be a set of n activation records in the run-time stack located in X memory and Y memory respectively where, in* A_{ik} *and* A_{jk} *for function k* $(0 \leq k \leq n-1)$, A_{Xk} *is one and* A_{Yk} *is the other. When the activation records are assigned in the run-time stacks for X and Y memory to minimize the difference between size of* A_X *and size of* A_Y*, we say that the two run-time stacks are* ***balanced*** *at given instant.*

If we have large enough on-chip memory to execute a whole program, value of balanced dual run-time stacks is disappeared. However, on-chip memory is limited, (Motorola DS56301 has 3k words for program memory, and 2k words for X and Y memory, respectively [3]), and in case that on-chip memory is fully utilized, we cannot avoid using off-chip memory. If we use off-chip memory, it is impossible to access simultaneously to each memory bank, and because extra clock cycle is required, an instruction cannot be executed in one machine cycle. Therefore, use of off-chip memory by unbalancing lead to performance degradation. The next figure shows an example to describe a difference of unbalanced and balanced dual run-time stacks.

Fig. 2 shows two kinds of dual run-time stacks. The left one is the worst case of dual run-time stacks and the right one is balanced dual run-time stacks. From this figure, we can estimate that run-time stack of X memory can exceed the extent of on-chip X memory, and new activation record 4 has to be located on the external memory although there is free area in on-chip Y memory. However, in the balanced dual run-time stacks generated by reassignment of activation records, new activation record 4 can be located on the on-chip memory, and there is no performance degradation as shown in Fig. 2. Therefore we can convince that balancing activation records between X memory and Y memory leads to enhance utilization of on-chip memory and increase performance of execution speed.

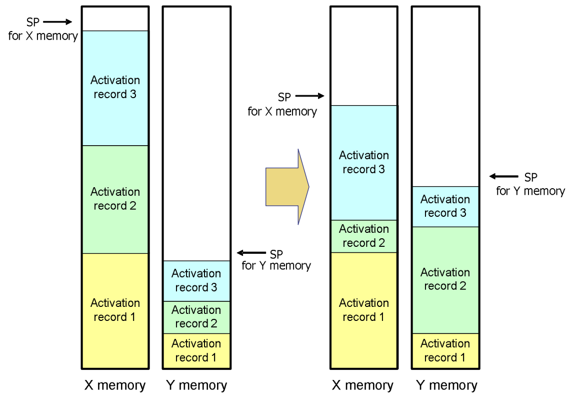


Fig. 2. Balanced dual run-time environment

3.2 Our Approach

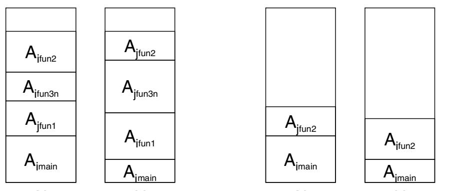
Before we start to present our approach to balance dual runtime stacks, we have to restrict the area of our consideration to enhance utilization of on-chip memory. We will explain with the example presented in Fig. 3. There are four functions, *main*, *fun1*, *fun2*, and *fun3*, and there are two paths to call function *fun2*; one is *main* → *fun1* → *fun3* → *fun2*, and the other is *main* → *fun2*, as shown in Fig. 3 (a). All activation records have to be assigned in X and Y memory like this: $AX = \{A_{imain}, A_{jfun1}, A_{ifun3}, A_{ifun2}\}$ and $A_Y = \{A_{jmain}, A_{ifun1}, A_{jfun3}, A_{jfun2}\}$ for the first path as shown in Fig. 3 (c), and $AX = \{A_{imain}, A_{jfun2}\}$ and $AY = \{A_{jmain}, A_{ifun2}\}$ for the second path as shown in Fig. 3 (d), respectively.

```
main()              fun1()
{                   {
   fun1();             fun3();
   fun2();          }
}
                    fun3()
fun2()              {
{                      fun2();
                    }
}
```

(a) C source code

	A_i	A_j
main	8	4
fun1	8	6
fun2	7	5
fun3	5	9

(b) Each activation record size



(c) Balanced Dual Run-time stacks (main -> fun1 -> fun3-> fun2) (d) Balanced Dual Run-time stacks (main -> fun2)

Fig. 3. Example of balanced dual run-time stacks

From the result, we can know that function *fun2* has to be assigned in different memory in order to be balanced the run-time stacks according to calling sequence. This problem can be resolved by function cloning, but it causes code size to increase. Therefore, we have to decide which one to be balanced in numerous calling sequences. We call this problem ***inter call chain balancing (Inter-CCB)*** which means relation of two calling sequences; *main* $\rightarrow$ *fun1* $\rightarrow$ *fun3* $\rightarrow$ *fun2* and *main* $\rightarrow$ *fun2* from Fig. 3. And we define ***intra call chain balancing (Intra-CCB)*** in similar manner which means relation of activation records in the calling sequence from root to leaf activation node. Thus we divide balancing problem for DDMB into *Intra-CCB* and *Inter-CCB* problem to approach effectively. In next section, we describe in more detail how *Intra-CCB* and *Inter-CCB* algorithms work.

4 Balanced Run-Time Environment

In this section, we discuss how run-time memory can be optimized in DDMB and what the benefits are, especially for run-time memory. First, we describe our *intra call chain balancing (Intra-CCB)* algorithm to optimize usage of run-time memory in a single call chain, and then we provide *inter call chain balancing (Inter-CCB)* algorithm to decide order of *Intra-CCB*. To describe easily our balancing algorithm, we divided a program into two cases; with and without recursive calls. And in this paper, we just deal with the program without recursive calls. Before describing our algorithm, we provide first a few basic concepts in next subsection.

4.1 Basic Concepts

To find a call chain to apply our *Intra-CCB* algorithm, we define extended activation tree first. We extend activation tree defined in [1] to support DDMB architecture.

Definition 2. *Let T = (V, E, W) be a weighted tree where V is a set of activations of procedures, E is a set of edges which mean procedure call relations, and W is a set of weights which mean increased size of X and Y activation records when child procedure is called. An edge* (V_i, V_j) *means procedure Vi calls* V_j*, and* $Wk = (W_{xk}, W_{yk})$ *means X memory increase* W_{xk} *size and Y memory increase* W_{yk} *size when the procedure is called. We call the weighted tree T to extended activation tree (EAT). Above defined EAT has pre-defined node, root node, which represents global/static area.*

Start address of two stacks are different depends on global and static variables in X and Y memory banks. To consider these different start addresses, we use a default *root* node. We imagine that this *root* node calls *main* function with weight pair which means memory usage used in X and Y memory banks for global and static variables. To explain our balancing algorithm for DDMB, we define *Balancing Factor*.

Definition 3. *The Balancing Factor (BF) is defined as ux - uy, where ux and uy are the amount used of the X and Y memory, respectively.*

We find that our definition allows us to more easily explain our algorithm. From Definition 3, if BF has a positive value, it indicates that the amount of used X memory is more than the amount used of Y memory. If BF is zero, because it indicates that the amount used of two memories are same, dual run-time stacks is balanced at the instant. If BF has a negative value, it indicates that the amount used of Y memory is more than the amount used of X memory. The larger BF, the less two stacks are unbalanced. Therefore, our algorithm finds the path first which has the largest BF, and then the path is tried to balance through swap of activation records in X and Y memory.

4.2 Intra Call Chain Balancing

The swap of the activation record in X and Y memory means that swappable fields in X and Y memory exchange each other. From an activation record, arguments, temporary variables, local variables, caller-save registers, and callee-save registers are swappable fields, and return address in X memory is an unswappable field. In our previous work [2], we first identify a *maximum spanning tree* (MST) of a *simultaneous reference graph* (SRG), and then X memory is assigned in even depth and Y memory in odd depth in this tree. Swap of an activation record means that X memory is assigned in odd depth and Y memory in even depth in that tree. Therefore swap of an activation record does not affect correctness of program. However, because return address has to be saved in a caller and restored in a callee, we fix the field in X memory. Although the status register is saved and restored in a caller, we fix the register in Y memory to consider a return address in X memory. Thus, when activation records are swapped, return address and status register are excluded.

The core of our balancing algorithm is described at from Line 7 to Line 27 of Fig. 4. Basic concept of our algorithm is a greedy algorithm to assign small activation to

```
Input: a call chain p
Output: a node          // root node of subgraph which has swapped activations
algorithm: IntraCCB
1  q = sort (p);                    // descending sort for each activation record of
2                                              // the found call chain
3  // Balancing the found path with descending order
4  A ← {};                                     // Set of assigned nodes
5  recal_node ← NULL;               // The node to recalculate BF
6  current_bf = 0;                         // Current BF
7  for all activation records $q_i$ in q do
8        if ($q_i$ is not an element of A)
9              if ((current_bf > 0) and (size(X$q_i$) > size(Y$q_i$)))
10                   swap(X$q_i$, Y$q_i$);
11                   if ((recal_node equals to NULL)
12                               or ($q_i$ is not descendent of recal_node))
13                         recal_node ← $q_i$;
14                   end if
15             end if
16             if ((current_bf < 0) and (size(X$q_i$) < size(Y$q_i$)))
17                   swap(X$q_i$, Y$q_i$);
18                   if ((recal_node equals to NULL)
19                               or ($q_i$ is not descendent of recal_node))
20                         recal_node ← $q_i$;
21                   end if
22             end if
23             A ← A U $q_i$;                          // To fix assignment
24       end if
26       current_bf = current_bf + bf_op(X$q_i$, Y$q_i$);
27 end do
28 return S;
end algorithm
```

Fig. 4. Intra call chain balancing algorithm

the larger size memory bank one after the other. In the initial state, because there is no assigned activation of X and Y memory banks, the first activations of the found call chain are assigned with the pre-assigned activations from our MST algorithm in the previous stage. From the second activations of the call chain, if the assigned words of X memory bank are more than them of Y memory bank, the more activation of the two is assigned to Y memory bank. Above sequence is performed iteratively for all activation nodes of the path.

We use a greedy approach including descending sort in order to balance the calling sequence. At Line 1 of Fig. 4, descending sort is performed to handle some larger activation first. If sorting is excluded from our algorithm, we might not be able to assign a memory bank optimally in order to use a minimum amount of run-time memory when the flow of control in a program corresponds to this call chain.

4.3 Inter Call Chain Balancing

From the example of Fig. 3, we can know that requirement of balancing is conflict. If there is a function having two callers, balancing of the activation record in one call chain can make effect the activation record in the other call chain.

Definition 4. *Let A is an activation record. If the number of caller of A is more than one, we call A* ***overlapped*** *activation record. The root node of Definition 3 and main of C language are default overlapped activation record. We also say a call chain not including even an overlapped record independent call chain, otherwise dependent call chain.*

From Definition 4, we can know that all call chains are dependent because they include at least root and main activation records. Therefore finding an optimal balancing from all call chains of the EAT is NP-complete problem. As described previously, because our main aim is to enhance utilization of on-chip memory, we attempt the balancing for dual run-time stacks at instant when it is maximized to difference of the sum of activation records between X memory and Y memory, that is, most unbalanced instant. Therefore we apply our balancing algorithm to the call chain which has the largest difference usage of X and Y run-time memory, and if we already assigned activation records to X or Y memory, we skip assignment of the

```
Input: EAT
Output: Balanced EAT
algorithm: InterCCB
1  E = {};                          // A set of assigned edges in EAT
2  L = findLeafNodes(EAT);          // A set of leaf nodes in EAT
3  calculate_BF(EAT);
4  while (L != 0)
5      l = find a node with maximum BF in L;
6      p ← path from root node to l;
7      S = IntraCCB(p);
8      recalc_BF_of_subtree(S);
9      L ← L – {l}
10 end while
end algorithm
```

Fig. 5. Inter call chain balancing algorithm

activation records. Our balancing algorithm is repeated iteratively until there is no change or all activation records are fixed. Figure 5 shows our balancing algorithm in more detail.

5 Experimental Result

To evaluate the performance of my algorithm for run-time memory optimization, we implemented the algorithm in our compiler and conducted experiments with IDCT, G721 encoder, and G721 decoder of MediaBench benchmark suites [4] on a commercial DSP, the Motorola DSP56300 [3]. The performance is measured in usage of run-time memory corresponding to procedure calls. In this section, we report the performance obtained in our experiments, and compare our results with other work.

5.1 Measurements of Memory Usages

We compared our run-time environment with fully static dual run-time environment used in SPAM compiler [7][8] to demonstrate the effectiveness of our CCB algorithm. Besides, by comparing with SPAM, we can analyze the pros and cons of our CCB approach as opposed to their fully static dual approach. To resolve an effect of memory bank assignment, we just extract fully static run-time environment from SPAM compiler, that is, we calculate static size of X and Y run-time stacks from result of our memory bank assignment. We convince that this experimental approach is fair to compare only usage of run-time memory.

Table 1. Benchmark results

	IDCT	G721 encoder	G721 decoder
Fully static dual approach	1165	421	500
CCB approach	1125	257	272

Table 1 shows result about benchmark programs. The value means total usage of run-time memory, that is, sum of X and Y run-time memory. In case of IDCT, stack area was saved 27.1% and total memory was saved 3.3% because this benchmark used 1025 words as global/static area. In g721 encoder benchmark program, stack area was saved 48% and total memory was saved 39% because this benchmark used 80 words as global/static area. And in g721 decoder benchmark program, stack area was saved 54% and total memory was saved 45.6% because this benchmark used 80 words as global/static area.

5.2 Memory Optimization

In this subsection, we compare maximum usage of run-time memory between the worst case and the case applied our CCB algorithm. Because IDCT benchmark program has large size of global variables, effect of CCB algorithm is relatively small. Therefore we experimented with two benchmark program; g721 encoder and decoder. In this experiment, we evaluated maximum memory usage in X and Y run-time memory whenever run-time stack was changed, that is, procedure is called and exited.

From our results, our CCB algorithm decrease 9% (G721 encoder) and 10% (G721 decoder) of run-time memory at instant of maximum memory usage.

6 Conclusions

Many DSP vendors provide a dual data memory bank system that allows applications to access two memory banks simultaneously. In previous work, we addressed a decoupled approach to exploit multiple memory bank architecture. In this paper, we proposed a call chain balancing algorithm to utilize on-chip memory efficiently, where Inter-CCB and Intra-CCB algorithms are performed. Inter-CCB algorithm selects a call chain to balance in EAT, and Intra-CCB algorithm performs balancing of activation records for the found call chain. Because balancing between X and Y memory usage means that maximum memory usage of X or Y memory bank is diminished, it will give more chance to utilize on-chip memory. The comparative analysis of the experiments revealed that our CCB algorithm achieves better results in usage of run-time memory than a previously described fully static run-time environment.

Acknowledgments. This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031), KRF contract D00191, the Korea Ministry of Information and Communication under Grant A1100-0501-0004, the Korea Ministry of Science and Technology(MoST) under Grant M103BY010004-05B2501-00411, IT R&D Project funded by Korean Ministry of Information and Communications, and nano IP/SoC promotion group of Seoul R&BD Program.

References

1. Aho, A. V., Sethi, R., AND Ullman, J. D.: Compilers -Principles, Techniques, and Tools. *Addison-Wesley Publishing Company* (1986)
2. Cho, J., Paek, Y., AND Whalley, D.: Fast Memory Bank Assignment for Fixed-Point Digital Processors. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 9, Issue 1, (2004) 52–74
3. Motorola Inc. http://www.motorola-dsp.com. DSP56301 User's Manual (1999)
4. Lee, C., Potkonjak, M., AND Mangione-Smith, W.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annaul IEEE/ACM Internation Symposium on Microarchitecture*, (1997) pages 330–335
5. LEUPERS, R. AND KOTTE, D.: Variable partitioning for dual memory bank DSPs. In *Proceedings of the IEEE International Conference on Acoustics Speech and Signal Processing*. (2001) 1121–1124.
6. Liem, C.: Retargetable Compilers for Embedded Core Processors. Kluwer Academic Publishers (1997)
7. Saghir, M. A. R., Chow, P., AND Lee, C. G.: Exploiting Dual Data-Memory Banks in Digital Signal Processors. *ACM SIGOPS Operating Systems*, (1996) 234–243.
8. Sudarsanam, A. AND Malik, S.: Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, Issue 2 (2000) 242–264