Exploring the Number of Register Windows in ASIP Synthesis

Vishal P. Bhatt*,
Synposys India
vishu@synopsys.com

M.Balakrishnan, Anshul Kumar
Department of Computer Science & Engineering
Indian Institute of Technology Delhi, New Delhi, India
{mbala,anshul}@cse.iitd.ac.in

Abstract

ASIPs (Application Specific Instruction Set Processors) are one of the key components of many embedded systems which are typically application specific. An ASIP can be defined by a set of architectural features, number of register windows being one of them. The work described here focuses on generating the transfer time penalties for some of the mediabench benchmark applications namely JPEG and MPEG encoder and decoder, for different number of register windows. The problem has been solved in two steps. First the "spills" for different number of windows were counted where a spill refers to the situation where a context switch cannot be accommodated in the register windows thereby adding the overhead of transferring some data to the memory. This part of the problem was solved by mapping it to the "regular language recognition problem". In the next step, actual time penalties for different system configurations, were computed. Here, a system configuration consists of the memory configuration, bus width and speed and processor cycle time. Thus, this work may also drive the design space exploration process. Results and expected performance gains by selecting different number of register windows is presented.

1 Introduction

An ASIP is a processor designed for a particular application or a set of applications. It exploits the special characteristics of the application(s) to meet the desired cost, performance and power requirements. An ASIP configuration consists of a set of architectural features which are decided by looking at the applications which are going to run on the ASIP. Therefore, *application analysis* is a key step in the process of ASIP design[4]. Another important step is determining the impact of the architectural choices on the desired parameters such as performance or power consumption. A simple way to do this is to *simulate* an ASIP configuration after mapping the given application on it. For example, Kin [3] uses a framework which consists of an ILP compiler, processor simulators, set of media applications and an architectural component selection algorithm to identify architectural parameters so as to meet low power design goals under area constraints for application specific media processors. However, this approach is quite time consuming and is not suitable for extensive exploration of the architecture space. On the other hand, exploration of architectures driven by performance *estimation* is much more efficient, for example in [6].

It is desirable that ASIP design space exploration should incorporate memory space exploration as well. However, only limited work has been reported in this direction. The memory space is explored in [7], in an embedded systems environment such that the minimum energy cache configuration can be determined if performance is the hard constraint or the minimum time cache configuration can be determined if energy is the hard constraint. Binh et al. [1] propose an optimization algorithm which partitions the application into hardware and software such that the performance of the designed ASIP is maximized under the constraint of chip area including RAM and ROM sizes.

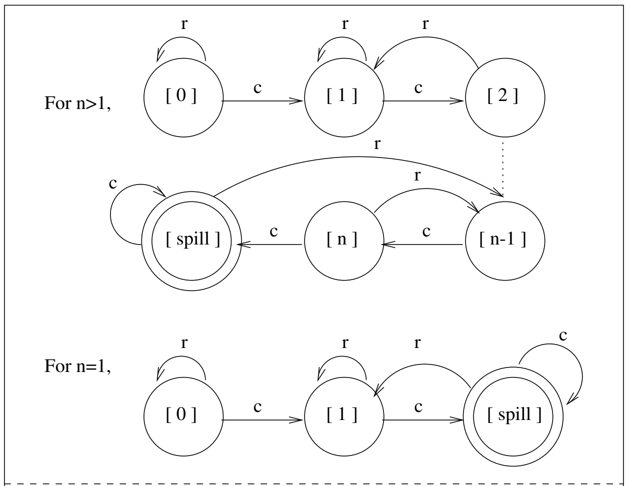
In this paper we focus on *estimating* the effects of different number of register windows on the performance of the system for different system configurations where a system configuration consists of the memory configuration, bus width and speed and the processor cycle time. The number of register windows in a processor affects the way *context switches* are carried out in the application. If all the windows are occupied when the application encounters a context switch, then one of the windows needs to be vacated for accommodating the new context switch. The content of one of the windows then needs to be stacked (transfered to the main memory). This event is usually referred to as *spill*. Spills are associated with time overhead incurred in transferring data to and from the memory due to lack of enough number of register windows. This overhead can be reduced by increasing the number of windows but that would in turn increase the cost. So, a typical decision would be some cost-performance trade-off. A method for estimating the cost and performance of the computing system for given number of register windows and given application is, therefore, very useful during ASIP synthesis. The work described here focuses on performance estimation.

2 Spill Count Computation

As stated earlier, a spill is an event where a context switch occurs when all the register windows of the processor are occupied. A typical context switch is the result of a function call. In such a situation one of the windows needs to be vacated so as to accommodate the new context. The time spent in vacating the window is the penalty paid for lack of enough number of windows. So, we need to get an estimate of the number of such events, or the *spill count*, which lead to time penalties. We need to estimate the *spill count* for the corresponding application for different values of the number of windows. Our goal is to do this without running or simulating the application repeatedly for different values of the number of windows or memory models. How this is achieved is described in the following subsections.

2.1 Basic Idea

As the spill count depends upon the sequence in which context switches occur while an application is running, we extract a trace of context switches from the application during the application analysis phase. This trace can then be analyzed for different values of the number of register windows to get the spill count. We represent this trace by a string A over the alphabet $\{c, r\}$ where c represents a call and r, a return. In this representation, the number of c's is equal to the number of r's (provided it is a terminating application and assuming that there are no interrupts). The spill counting problem can now be viewed as that of defining a function which maps this string A and a given number of windows $\mathbf{n}$ to a number giving the spill count. As shown in the next subsection, this function is simply a *deterministic finite automaton* (DFA) designed for the given $\mathbf{n}$. Therefore, spill count for different values of the number of windows can be obtained by feeding the same string A to the DFAs for different values of $\mathbf{n}$.



n = Number of register windows, c = Function call, r = Function return

<Q, S, [0], F, A>

where,

Q = Set of states = {[0], [1], [2]..........[n], [spill]}

S = Alphabet = {c,r}

[0] = Start state, [spill] = Spill state

F = Next state function, **Q x S -> Q**

A = Set of accepting states = {[spill]}

Figure 1: DFA for Computing Spill Counts

2.2 DFA for Spill Count

Figure 1 gives the DFAs for n number of windows (where $n > 1$) and for single window. Here, a non-accepting state $[i]$ represents the state of the system when i number of windows are occupied. The accepting state $[\mathit{spill}]$ represents the *spill state*. Essentially these DFAs help us in keeping the "snapshot" of the register windows.

The following points help in understanding the functioning of the DFAs:

- The DFA is fed the string representations of applications with single control thread only i.e. it doesn't consider multithreaded or multiprocess applications.
- The strings fed to the DFA must start with a c which corresponds to the call to the main function.
- Every input c takes the DFA from state $[i]$ to the state $[i+1]$ for $i < n$. The DFA starts in state $[0]$ from where it reaches to $[1]$ corresponding to the call to the main function. The state $[1]$ implies that one window is occupied. It is occupied by the main function (initially). After main, every function call occupies a new window if it gets a vacant one. So, with every c the number of windows

occupied gets incremented by one. This is what takes the system from the state $[i]$ to $[i+1]$.

- When the system reaches the spill state, all the windows being exhausted, one of the windows is vacated. The new function call occupies this vacated window. So, the total number of windows occupied doesn't change. We also get a spill since one of the windows needs to be vacated. So, if a function call is made when in the spill state, the state of the system doesn't change.

- When we encounter a return from the spill state, a window gets vacated. As stated earlier, in the spill state, all of the n windows are occupied. So, if one of these gets vacated, the system has n-1 non-vacant windows which refers to the state $[n-1]$. This is why an r from the state [*spill*] takes the system to the state $[n-1]$.

- Whenever a function returns, a window gets vacated. So, except for the spill state and the state [1], a return takes the system from the state $[i]$ to $[i-1]$. When the system is in the state [1], after the function returns either the parent function is brought to the registers from the memory or the application terminates. If the parent function is brought to the registers, we again have one non-vacant window. Essentially, as long as the application is running, atleast one window remains occupied. This is why a function return from the state [1] takes the system back to the state [1].

Note here that the starting state [0] will never receive the input r, but a transition labeled r from state [0] has been included to make the DFA *completely specified.*

2.3 Extracting the Call/Return Trace

We can obtain the sequence of calls and returns in an application by running the code with some instrumentation statements inserted in it. The instrumentation statements can be inserted in a single pass over the program. These statements are as follows.

1. **vb_fsm('c');**	:	Inserted before a function call. Inserts the character **c** in the trace string.
2. **vb_fsm('r');**	:	Inserted after a function call. Inserts the character **r** in the trace string.

3 Memory Access Models

Memory organization plays an important role in the design of application-specific systems, particularly those which access large amounts of data. Studies have shown that in many Digital Signal Processing applications, the area occupied and power consumed by the memory subsystems is upto ten times that of datapath[2], making memory a critical component of the design. One important difference between the role of the memory in embedded systems and in general-purpose processors is that, since there is only a single application in an embedded system, the memory structure is configurable, and can be tuned to suit the requirements of the given application. For example, the traditional memory hierarchy structure can be modified[5] in the context of embedded systems resulting in improved performance. Further while general-purpose systems are typically concerned only with the performance-related aspects of the memory systems, embedded systems have a more diverse objective function, involving area, performance and power consumption.

The design space to be explored while designing the memory subsystem is considerably huge. There are many characteristics of the memory which can be varied to obtain the different configurations. For example, *interleaved/non-interleaved* main memory, *burst/non-burst mode* of data transfer from the main memory, *unified/split* cache, policies like *CBWA/CBNWA/WTWA/WTNWA*, etc. As a part of the work discussed here, an attempt has been made to study the different possible memory configurations and enumerate the memory access time formulae of some of these [8]. For given memory configuration, these access time formulae give us the *average access time per word.* Further, for every *spill* we need to transfer a fixed number of words to (and from) the memory. This number depends on the register window size and the bus width. Suppose we have 32-bit register windows and 16-bit bus. Then, for every spill we need to transfer two words to and from the memory which leads to a penalty of four words. So, the spill count can give us the penalty for different number of windows in terms of number of words. Given this *word penalty* and the *average access time per word* we can calculate the *time penalty.* Please refer [8] for the list of access time formulae.
In general, processor and memory design need to go parallel to each other. As shown in the flow diagram (Figure 2), ideally it should be possible to decide the memory model of an application specific system by looking at the application. Some parameters can be

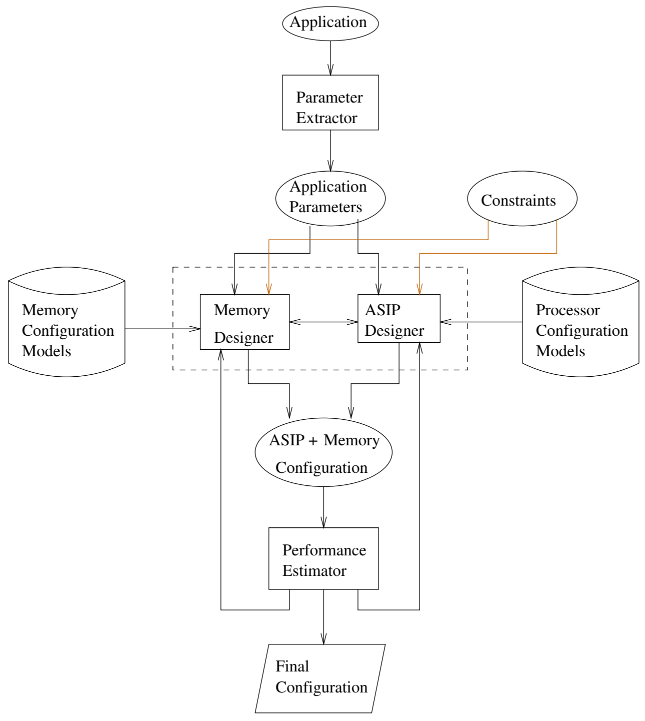


Figure 2: Flow Diagram ASIP and Memory design

extracted from the application which can further help in deciding the memory and the processor configuration. For instance, suppose that we want to decide the cache line size. If the application has a large usage of arrays then, if our line is big enough to accommodate the most frequently accessed array then we can expect the hit ratio to reduce. This is how the information extracted from the application can be utilized for deciding the system configuration.

Among the options considered while modeling memory include *Copy back write allocate* (CBWA) or *Write through no write allocate* (WTNWA), presence or absence of *Dirty line buffer* (DLB) and *Write through buffer* (WTB) and *burst or non-burst data transfer mode* (DTM).

4 Results

Using the methodology defined earlier, the spill counts for four mediabench applications namely *JPEG decoder*, *JPEG encoder*, *MPEG decoder* and *MPEG encoder* were generated. The results for *MPEG decoder* and *MPEG encoder* are included in figures 3 and 4 respectively. D1, D2, D3 and D4 refer to *m2vstreams* of *head, flowgard, track 12* and *smoker* respectively. These figures show the variation of the spill counts with

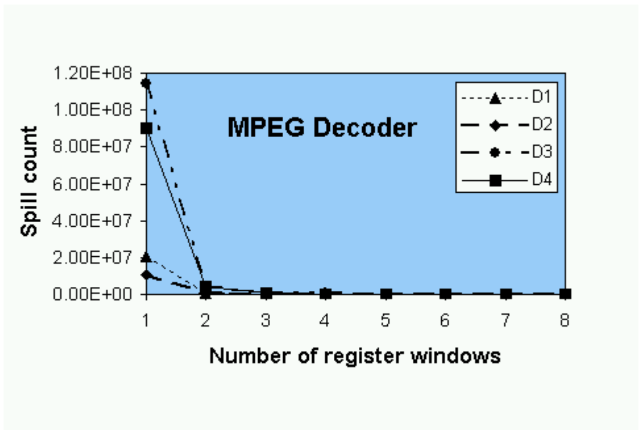


Figure 3: Spill Counts for MPEG Decoder

the number of register windows. The spill count variation gives us an idea of the way function calls are distributed across the application. It was observed in all the four considered applications that there is a drastic fall in the number of spills as we increase the number of register windows from one to two. This means that there are very few instances in the application when two calls are made consecutively as compared to the total number of calls. Compare these results with those obtained for the *factorial* application, shown in figure 5. For such applications which are dominated by recursive function calls, the spill counts are expected to decrease slowly with the number of register windows. The reason is that most of the calls in such an application will be made consecutively. So, when we increase the number of register windows by one, it generally doesn't save more than one call.

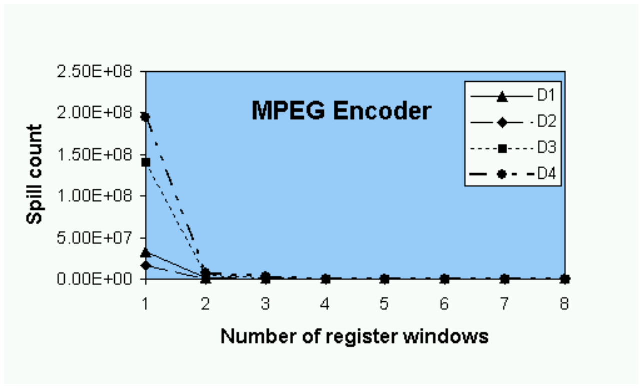


Figure 4: Spill Counts for MPEG Encoder

So as to get the estimate of the effect of the number of register windows on the performance of the system, we need to extend these spill count variation results as follows:

- Select a memory configuration and obtain the average access time per word for this configuration.
- Obtain the time penalty variations from the spill count variations and this average access time per word.
- Get an estimate of the total execution time of the application. In the work described here, this has been done by estimating:
  - total number of branch operations
  - total number of loads/stores
  - total number of operations

  in the application. Then, considering four cycles for every branch operation, two cycles for every load/store operation and one cycle for the rest of the operations, an estimate of the total execution time for the application has been obtained. The motivation behind including the estimate of the total execution time in the analysis is that we need to compare the time penalties with the total execution time.

Incorporating the total execution time estimate, the performance curves for six different system configurations have been generated. These are shown in figure 6. Here, results of three different memory models out of 16 have been included.

1. No cache - model number **0**
2. CBWA, Wraparound load, non-burst DTM - model number **3**
3. WTNWA, WTB present, burst DTM, interleaved memory - model number **15**

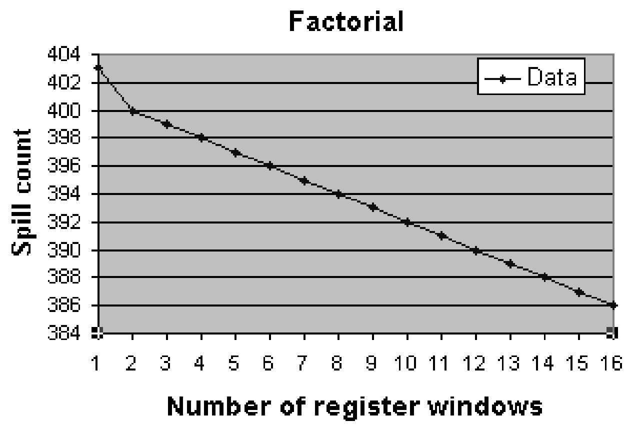


Figure 5: Spill Counts for the factorial application

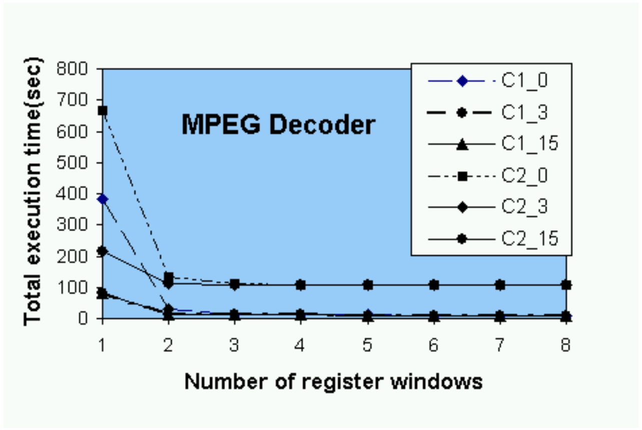


Figure 6: Performance curves for MPEG Decoder

These memory models are considered with the following two processor configurations; namely C1 and C2.

1. {200 MHz processor, 100 MHz 16-bit bus, 20 ns cache, 200-150 ns MM} - C1
2. {20 MHz processor, 10 MHz 16-bit bus, 30 ns cache, 300-250 ns MM} - C2

In figure 6, a configuration, *Ci_j* refers to config number *i* and memory model number *j*.

While generating the curves shown in figure 6, the moving picture file fed as input to the mpeg decoder is made up of 121 frames.Suppose that the mpeg decoder has to decode 10 frames per second which means the total execution time should not exceed 12.1 seconds (this defines the performance constraint on the application). If we take config number **1**, for a memory with no cache (memory model number **0**) we have an estimated total execution time of 12.14 sec for four windows and around 10.98 sec for five windows. Similarly, for the same config number and memory model number **3**, we have an estimated total execution time of around 13.96 seconds for two windows and 11.41 seconds for three windows. This data has been shown in table 1. This analysis, makes us conclude that instead of having a cache in our system, we may be able to do with no cache and five windows as that satisfies our constraint. Note here that this prediction ignores the effect of removing the cache on the performance of the rest of the system. If we decide to keep the cache (which means memory model number **3** or **15**), then we may decide to settle with three register windows. Instead, had the constraint on the decode rate been 8 frames/sec which means a total execution time of less than 15.125 seconds, we would have gone for two windows for memory model numbers **3** and **15** and three

Table 1: Result Table

Memory Model	Estimated Total Execution Time							
	1	2	3	4	5	6	7	8
0	384.74 sec	28.39 sec	14.84 sec **	12.14 sec	10.98 sec *	10.62 sec	10.62 sec	10.62 sec
3	80.78 sec	13.96 sec **	11.41 sec *	10.91 sec	10.69 sec	10.62 sec	10.62 sec	10.62 sec
15	80.76 sec	13.95 sec **	11.41 sec *	10.91 sec	10.69 sec	10.62 sec	10.62 sec	10.62 sec

* satisfy the constraint: *Decode rate of 10 frames/sec*

* satisfy the constraint: *Decode rate of 8 frames/sec*

windows for no cache (model number **0**).

5 Conclusions and Future Work

A methodology for predicting the time penalty for given application and given number of register windows has been proposed. In all the media bench applications considered, the results for several memory models show that the time penalty reduces drastically on increasing the number of windows from 1 to 2. The present work was carried out with the following simplifications. It is proposed to extend this work by removing these limitations.

- Presently, multithreaded applications have not been considered.
- Interrupts have been ignored.
- It has been assumed that all the variables of a function are getting accommodated into the windows. So, the penalty estimates can be low at many places. Taking into account, the penalty due to lack of enough number of registers in the individual windows, we can get more accurate estimates.
- The work needs to be integrated with the area and performance models of register windows of different sizes.

Acknowledgements This work was partially supported by the Embedded Systems Project sponsored by Naval Research Board, Govt. of India.

References

[1] Binh N.N., Takeuchi Y., Imai M. A performance maximisation algorithm to design asips under the constraint of chip area including ram and rom sizes. In *Proceedings of the Asia and South Pacific Design Automation Conference 1998 (ASP-DAC'98)*, pages 367–372, February 1998.

[2] F. Balasa, F. Catthoor, and H.D. Man. Background memory area estimation for multidimensional signal processing systems. *IEEE Trans. on VLSI Systems*, 3(2):157–172, June 1995.

[3] Kin J., Chunho Lee, Mangione Smith W.H., Potkonjak M. Power efficient mediaprocessors: design space exploration. In *Proceedings of the 36th Design Automation Conference*, June 1999.

[4] Manoj Kumar Jain, M.Balakrishnan, Anshul Kumar. Asip design methodologies: Survey and issues. In *Proceedings of the 14th IEEE International Conference on VLSI Design*, January 2001, India.

[5] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip, Optimisations and Exploration.* 1999.

[6] T.V.Gupta, P. Sharma, M.Balakrishnan, S.Malik. Processor evaluation in an embedded systems design environment. In *Proceedings of the 13th IEEE International Conference on VLSI Design*, pages 98–103, January 2000, India.

[7] Chaitali Chakrabarti, Wen-Tsong Shiue. Memory exploration for low power embedded systems. In *Proceedings of the 36th Design Automation Conference*, June 1999.

[8] Vishal P. Bhatt. Register Window Analysis in ASIPs. M.Tech. Thesis, Dept. of Computer Science & Engg., Indian Institute of Technology Delhi, April 2001.