Process Versions in Rapid Prototyping*

Achim Österling and Rolf Ernst

Institut für Datenverarbeitungsanlagen
Technical University of Braunschweig
Germany

Abstract

Rapid prototyping in general requires the mapping of a system model to a hardware architecture which is different from the design target architecture. To take into account easier function reuse or late specification changes both system designs should be developed in parallel. Different architectures, however, require different process adaptations and high–level optimizations. Therefore, several versions of process structures and high–level optimizations must be provided for the different architectures and component alternatives. In previous approaches, only one input version can be used which enforces design iterations over different versions of the input description.

The main focus of the paper is to show the necessity for introducing design versions of the process structure in rapid prototyping. The user predefines possible versions of the process structure, high–level optimizations, and process segment granularity. These versions shall be used in a system scheduling and allocation process. Furthermore, a suitable design environment is introduced.

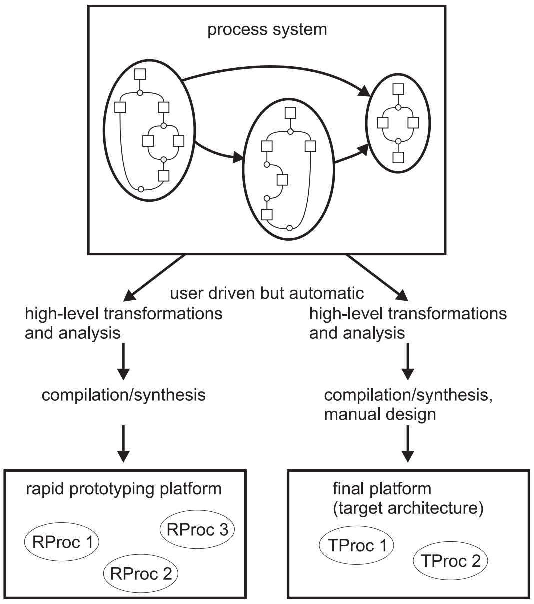

Figure 1: Outline of a rapid prototyping environment

1 Introduction and related work

Rapid prototyping in general requires the mapping of a system model to a hardware architecture which is different from the design target architecture. In this paper, we assume complex systems with reactive and transformative parts which shall be implemented on a rapid prototyping architecture consisting of several processors and FPGAs. We will look at the software part, only.

The rapid prototyping architecture is usually more complex and uses more powerful components than the target architecture where cost is a crucial factor. The different architectures typically require different high–level transformations for prototype and final design and they often use different design tools. Fig. 1 shows the principle. High–level transformations include simple operations such as loop unrolling, more complex operations such as loop tiling or restructuring for subword parallelism and even changing the process structure when a single process of the target architecture must be mapped to several processors of the rapid prototyping architecture. This paper will give a number of examples where optimizations show very different effects on rather similar processor architectures. There is an increasing set of techniques for automated high–level transformations, such as offered by the SUIF compiler system [11], but manual interaction is still required, as in the case of subword parallelism or complex DSP memory architectures. The resulting system models can have very different structures for different architectures.

Instead of first structuring the whole system to fit the rapid prototyping architecture and then restructuring the system for the design target architecture, it is more efficient to keep both versions of the system model in parallel. If part of the system is reused without modification, one obviously saves the time consuming restructuring step. But, even if the system is slightly modified for reuse, one could save time by replaying the optimization and adaptation steps, in particular in case of the rapid prototype which relies on a largely automated design process. The same argument holds for late and incremental changes of the specification.

For that purpose, the versions should contain the source code and the respective optimization options or scripts to

*This work is supported by the Deutsche Forschungsgemeinschaft.

be applied to a specific part of the system. Incremental user driven changes in both model structures or an incremental replacement of parts of the prototype by target system components may lead to conflicting optimizations requiring a sophisticated version management. In case of a heterogeneous rapid prototyping or design target architecture, versions can even be used for one architecture to exploit different design alternatives for process parts to take advantage of the different properties of the various components of the architecture.

Usually, the design team will be expert in the target system architecture and should not be bothered by the intrinsics of the rapid prototyping architecture. In case of more complex architectures, the design tools should give hints of what optimizations would be favorable, if they cannot be applied automatically. This is a second problem which this paper will look at.

However, the support of alternatives for process parts (which we call *process versions)* results in two problems. First, process versions require a variable granularity size of the elements used during the high–level transformation and synthesis steps to better utilize the architecture components (*AC*). In previous approaches, the size of these function elements ranges from processes in the input language [1] down to functions [12] and basic blocks or elementary threads [5, 6, 8], in ASIP design even down to individual statements [13]. The optimization potential for a system scheduling and allocation process (for brevity, we will call it mapping process) rises with finer granularity (small elements), but the larger design space increases optimization cost and limits the system size or requires the use of simpler heuristics.

A uniform granularity as it is used in most approaches is in general not justified by the application. Often, parts of the system clearly belong together, e.g. due to high internal communication, while other parts allow many implementation alternatives. Moreover, a coarse granularity (e.g. process level) may inhibit the efficient exploitation of properties from both reactive and transformative parts inside one process whereas a fine granularity (e.g. basic block level) may inhibit the application of optimizations which require a coarser granularity such as loop transformations. Therefore, some authors have proposed clustering strategies, either manual [9] or based on some "closeness" criteria [2, 3] originally used in high–level synthesis [4] which are applied on different levels of granularity. In [7], it has been demonstrated that dynamic adaptation of granularity can be useful to obtain better partitioning results. Clustering, however, must be introduced carefully because it increases the design space since a cluster may completely contain clusters on deeper nesting levels such that a function element is member of a lot of clusters. In [7], e.g., clustering is, therefore, constrained to the control hierarchy of the input processes.

In contrast to high–level synthesis which uses rather small elements such as function units and registers, rapid prototyping uses large processor and memory components. A human designer is able to effectively reduce the design space according to the suitability of a function element or a large data object for a certain component. This knowledge influences component selection, granularity, and mapping of function elements to components. Moreover, as mentioned above, the designer would adapt high–level optimizations such as function inlining, loop tiling, loop fusion, etc. to the selected suitable architectures.

The second problem implied by supporting process versions is the potential overlap of function elements. Overlapping function elements are obtained if the designer rates parts of the function elements suitable for more than one architecture. Overlapping function elements must be considered in a design environment. First, during high–level transformation and analysis, all function elements must be represented and stored such that a designer can access the different versions of a process. Second, during the mapping process, versions must be considered in so far as each atomic unit of the system specification (e.g. a basic block) must be mapped only once to any AC to obtain a valid system partition. Furthermore, it must be determined whether the mapping process found a valid and complete system partition and, therefore, can terminate.

This paper first gives examples for the adversary effects of high-level transformations. We will show that a transformation increasing the performance of an architecture will reduce the performance of another one. We will then present a design environment and briefly sketch a design representation suitable for parallel design with different process structures, thereby controlling design consistency.

2 Problem definition and results

We will start with a simple example. Suppose, there is a (signal processing) process with three fixed size loops, as shown in fig. 2. It might be useful to partially unroll and fuse the three loops for high–level synthesis or standard microprocessor implementation. The result is an increased program size at higher execution speed. This is a high–level transformation which optimizes the process under the condition that all three loops are mapped to the same component. If not, then this optimization was a bad idea. As a consequence, the three loops should form a single function element in this specific version.

Now, take a DSP with specialized "zero–overhead" loop instructions. There is negligible performance gain by partial loop unrolling, but there might be a very small program cache, as in the popular ADSP2106x, which will cause drastic cache miss overhead. In the version for this processor, there will be neither loop unrolling nor loop fusion and each of the nested loops will be a function element.

Assume another processor with a constrained data memory size or data cache or a smaller memory component in a larger architecture. Due to space constraints, it is not possible to store all arrays A to I in the memory (see fig. 2). On the other hand, loop fusion and partial unrolling are highly advantageous because of a large internal register set (or fast internal memory), where the designer can

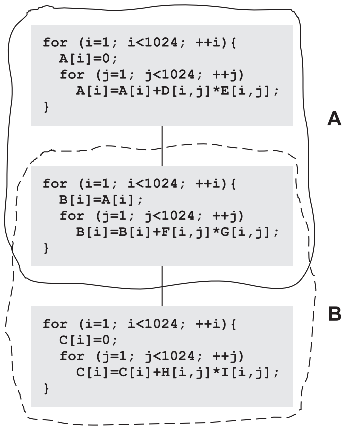

Figure 2: A capacity problem

make use of memory disambiguation to minimize memory accesses. In this case, either the first two loops or the last two loops fit on the component with its internal memory. As fig. 2 shows, function element A overlaps with function element B and, therefore, they are mutually exclusive and cannot be mapped to processors at the same time. Examples for components where such situations occur are high–performance video DSPs, such as the TMS320C80. Even in current high–level synthesis systems we will face the same situation if there are memory size constraints.

The example should be understood as a small part of a larger system, where there might be many such alternatives which must be treated by optimization. Note that, in this scenario, the designer is still responsible for defining these optimization alternatives, but introducing versions allows us to exploit the design space of both the rapid prototype system architecture and the final system architecture in parallel, while in clustering this requires iterations because only one version can be chosen at a time.

This shows that only having a single version of a function element may have negative impact on the system performance depending on where the mapping step places a function element. A positive impact can be achieved when a function element had been optimized for the AC it will later be mapped to. Since the mapping step is executed after high–level optimization the mapping step must have available different versions of function elements to take advantage of architecture–specific optimizations when mapping the function element to an appropriate AC. Moreover, by providing several versions we do not constrain the design space of the mapping process which may also consider system properties such as rate or latency constraints or power dissipation constraints which are not taken into account during high–level optimization and granularity adaptation.

Thus, the quality of the mapping result increases when the suitability of a function element for an architecture is considered. This entails that a function element can be optimized for this particular architecture since different and overlapping process versions are supported in our approach.

Note that, by supporting overlapping function elements, we avoid to constrain the optimization potential of atomic units of a process (e.g. a basic block) because an atomic unit may belong to several function elements and, therefore, may be optimized for several architectures. This optimization is done considering the whole function element, it is not limited to a single atomic unit. Finally, it is up to the mapping process to determine which function element should be mapped to which AC. Since this mapping is done considering the user classification, all atomic units of the function element are optimized for the selected AC.

Another problem is that some programs simply do not compile for certain architectures. In embedded real–time systems, where it is crucial to meet timing constraints, one has to consider memory hierarchy to take advantage of spatial and temporal locality by transforming code and data to speed up the system. The architectures which are used in these systems mostly only dispose of a small amount of internal fast (cache–)memory, so high level optimizations may lead to misses when trying to access data or instructions in the internal fast memory. This may result from the increase in code size (e.g. by loop unrolling) or in the lack of spatial locality by not applying optimizations such as array merging, loop interchange, loop fusion, or loop tiling.

Due to the different sizes of internal memory or, in general, the specialization of architectures such as digital signal processors a single function element will often not compile on all available architectures. As a consequence, the design space of the mapping process is constrained which may result in a non–feasible system design. This means that the availability of process versions is a requirement for successful system design in a heterogeneous processing environment.

Now, we present some more results to underpin our statements. We try to map a benchmark to three different architectures. The first architecture, the TMS320C62x, is a VLIW style digital signal processor running at 166 MHz. Due to its VLIW style core it is predestined for exploiting instruction level parallelism and for the application of optimizations such as loop pipelining. It has an internal data memory size of 64 kByte. The second processor is the TriMedia TM–1 which is also a VLIW style digital signal processor. The TriMedia has a 32 kByte instruction cache and a 16 kByte data cache, both are 8–way associative. The processor runs at 100 MHz. The third processor is the StrongARM. This processor has been optimized for low power consumption and, therefore, is predestined for the usage in portable electronic systems. It has a simple system architecture. The StrongARM has an instruction and data cache. Both have a size of 16 kByte and are 32–way associative. The processor runs at 160 MHz.

We emphasize the internal memory sizes because we want to deal with a problem mentioned above: the different sizes of internal memory and associated optimizations. As benchmark we choose a matrix multiplication algorithm. The program is shown in fig. 3. First, we multiply two 64×64 arrays. Table 5 shows the results for this benchmark. It takes 58 ms on the C62x, 49 ms on the TriMedia, and 461 ms on the StrongARM to execute the program. However,

```
#define BOUND 64

int i, j, k, A[BOUND][BOUND],
    B[BOUND][BOUND], C[BOUND][BOUND];

void main (void)
{
  for (i = 0; i < BOUND; ++i)
    for (j = 0; j < BOUND; ++j)
      for (k = 0; k < BOUND; ++k)
        C[i][j] = C[i][j]+A[i][k]*B[k][j];
}
```

Figure 3: The original program

```
#define BOUND 64
#define STEP 16
#define min(x,y) ((x)<(y)?(x):(y))

int i, j, k, i1, j1, k1, imin, jmin, kmin,
    A[BOUND][BOUND], B[BOUND][BOUND], C[BOUND][BOUND];

void main (void)
{
  for (i = 0; i < BOUND; i += STEP)
    for (j = 0; j < BOUND; j += STEP)
      for (k = 0; k < BOUND; k += STEP)
      {
        imin = min(BOUND,i+STEP);
        jmin = min(BOUND,j+STEP);
        kmin = min(BOUND,k+STEP);
        for (i1 = i; i1 < imin; ++i1)
          for (j1 = j; j1 < jmin; ++j1)
            for (k1 = k; k1 < kmin; ++k1)
              C[i1][j1] = C[i1][j1]+A[i1][k1]*B[k1][j1];
      }
}
```

Figure 4: The tiled program

to reduce the number of cache misses for the StrongARM and the TriMedia, we tile the loops. We choose tiles of size 16×16 such that all necessary data fits into the cache. The tiled program is shown in fig. 4. Now, it takes 274 ms on the StrongARM which is a speedup of 68% compared to the original program. It takes 69 ms to execute the tiled program on the C62x which is a performance degradation compared to the original program. This is due to the increased number of loops in the tiled program, i.e. the increased number of initializations and the min calculation. Here, the C62x cannot take advantage of loop tiling because the data already fits into the internal memory for the original program. Therefore, executing the tiled version on the StrongARM leads to a performance improvement while executing it on the C62x leads to a performance degradation compared to the original program. On the TriMedia, we only obtain a slight performance improvement of 49 ms vs. 48 ms because the data cache hit rate of the original program is already 98% (100% in the tiled program).

Next, we multiply two 128×128 arrays. For the C62x, we do not obtain a solution because the compiler cannot allocate the required memory. We need 196608 byte for the 3 arrays but only 64 kByte are available. Therefore, it would be necessary to implicitly code the data transfer from external memory to internal memory via the DMA unit if the program is to be executed on the C62x. For the StrongARM, loop tiling improves the system performance again. We obtain a speedup of 84%. For the TriMedia, we obtain a greater performance improvement compared to the multiplication of the 64×64 arrays because the data cache hit rate of the original program is only 89% while it is 100% for the tiled program. Here, it takes 575 ms to execute the original program and 382 ms to execute the tiled program.

Note that, for the StrongARM and the TriMedia, we need external memory in both cases. However, the cache hardware is responsible for fetching the required data in case of a cache miss. For the C62x, it is up to the programmer to ensure that the required data can be accessed. Therefore, it is not sufficient to apply high–level transformations and hope that the result will compile on all available architectures. In fact, the target architecture must be considered.

The results described above have been obtained without further compiler optimization to demonstrate the effect of loop tiling. For the results depicted in fig. 6 we select the best optimizations the compilers can offer. In our case, we observe two effects. First, the compiler optimizations can not reverse the effects shown in fig. 5, i.e. tiling still leads to a performance degradation in case of the C62x while it leads to a performance improvement in the other cases. Second, by applying the high–level optimization before the compiler optimizations results in a better system performance than not applying the high–level optimization before the compiler optimizations, e.g., for the StrongARM, $\frac{461\text{ ms}}{241\text{ ms}} < \frac{274\text{ ms}}{87\text{ ms}}$, i.e. the speedup is larger in the latter case. Again, this is not the case for the C62x where loop tiling does not lead to a performance improvement.

	C62x	TriMedia	StrongARM
64×64	58 ms	49 ms	461 ms
64×64 tiled	69 ms	48 ms	274 ms
128×128	—	575 ms	4040 ms
128×128 tiled	—	382 ms	2193 ms

Figure 5: Results for the benchmark without compiler optimization

	C62x	TriMedia	StrongARM
64×64	6 ms	15 ms	241 ms
64×64 tiled	11 ms	11 ms	87 ms
128×128	—	359 ms	2542 ms
128×128 tiled	—	91 ms	698 ms

Figure 6: Results for the benchmark with compiler optimization

3 A design environment

3.1 Introduction

This section describes the problems which emerge when supporting overlapping process versions in the design process (see also fig. 2). There are, at first glance, two major cases where overlapping process versions must be considered during the design of a system.

First, the process versions must be considered during user–driven high–level optimization and granularity adaptation. We devised a method to uniquely address these overlapping function elements and to store them.

Second, overlapping function elements must be considered during the mapping process since the validity of a function element mapping depends on whether any element already mapped to any AC overlaps with the currently selected function element. In this case, mapping the current function element would be an invalid step. These feasibility checks include the determination of function element overlap and containment. Moreover, to further increase the quality of the mapping result, determination of mutual exclusion of function elements will be included, too. Finally, it must be determined whether the mapping process found a valid and complete system partition and, therefore, can terminate.

In [10], we introduce data structures and a formal notation for the representation of overlapping function elements. There, we show how the feasibility checks described above can be realized efficiently. The design environment introduced in the next section is also based on these data structures and the notation.

3.2 Design environment architecture

In the design environment, the user can determine the granularity of function elements, specify high–level optimizations, and classify the function element for a particular architecture, thereby creating process versions, to enable a subsequent mapping process to work with process versions. The mapping process which maps the function elements to ACs should consider these classifications, thereby improving the function element architecture matching.

For rapid prototyping, the user adapts the granularities of the function elements and selects high–level optimizations for them according to the rapid prototyping architecture and design target architecture, resp. Note that, in case of a heterogeneous rapid prototyping or design target architecture, there is an additional dimension the user has to consider. Now, he has to select the function element granularity, the high–level optimizations, **and** the architecture component he wants to optimize the function element for.

To guide the user through these design steps a graphical user interface (GUI) shall be provided. In this GUI a graphical representation of the atomic units (in our case basic blocks) of a process (system) and their properties are displayed (see fig. 7). Both architecture independent properties and, upon user request, architecture dependent properties have to be considered. Interesting properties may be instruction level parallelism, the communication density among blocks, applicability of zero overhead loops, presence of potential three–address–instructions, data size and locality when accessing arrays to reduce the number of internal memory/cache misses, as well as the presence of floating point operations, required bit sizes, and required functional units to further reduce the set of eligible architectures and, thereby, the run time of a subsequent mapping process.

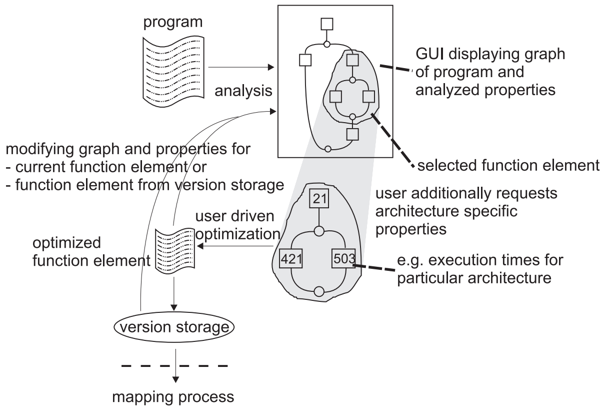

Figure 7: A design environment

First, the user determines the granularity of the function elements and selects high–level optimizations according to the architecture component in question and taking into account the displayed properties. These optimizations will automatically be applied to the function elements. Granularity selection and high–level optimization is an iterative procedure.

After function element selection and high–level optimization, an interface to a mapping process will be generated based on the user's classification. The task of the mapping process is to map the user determined function elements to ACs. Our approach, which only considers the internal properties of a process, separates process internal information from process external information such as latency or rate constraints which is taken into account by the mapping process usually based on an iterative improvement optimization strategy. Therefore, our approach is neither tied to a particular process model with its different constraints nor to a particular optimization strategy. Besides the process external information the user's function element classification shall be considered when mapping the function elements to the ACs. It is not valid to have overlapping function elements mapped to ACs at the same time. So, it is up to the mapping process to ensure that this case does not arise. The generated interface supports the mapping process by providing the feasibility checks for a potential mapping. Moreover, this process must be able to check if a valid and complete partition (function element to architecture mapping) of the process system was found such that the mapping process can terminate. As mentioned above, [10] shows how the feasibility checks and the termination condition can be realized.

3.3 Version storage and access

In this section we introduce our version storage and access concept. One has to note that a function element's number may not be unique when storing and accessing it. This is due to the fact that a function element may have been optimized for several architectures. Moreover, several op-

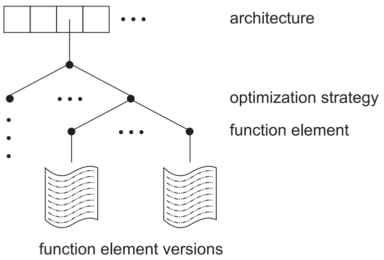

Figure 8: Version storage data structure

timization strategies may have been applied to the function element for each of these architectures.

How could the user access an already stored function element? By clicking on a block in the GUI all function elements the current block belongs to are displayed. There, the user can select the function element he really wants to access by specifying the architecture name, the optimization strategy, and the function element name.

Therefore, versions can be stored in a dictionary indexed by the architecture name the function element has been optimized for (see fig. 8). Since more than one optimization strategy could have been applied to the function element for the current architecture, we have another layer of entries indicating the used optimizations. The last layer contains the optimized function elements.

In reality, the implementation of the version storage will not store the modified function elements in the data structure but in a file system controlled by a revision control system to be able to revert to a previous version if a transformation sequence did not yield the expected result or to replay a transformation sequence. The dictionary will just contain references to identify the requested function element. A more detailed description of the version storage part of the design environment will be given in a future paper.

4 Conclusion

We have shown that high–level optimization and granularity adaptation lead to process versions with different structures. These different structures contain overlapping function elements which can lead to incompatibilities between versions. We have shown how an unsuitable application of high–level optimizations can lead to a performance degradation for the entire system. For rapid prototyping, it is desirable to develop both the rapid prototyping design and the final design in parallel to take into account easier function reuse or late specification changes. Therefore, several versions of process structures and high–level optimizations must be provided for the different architectures and component alternatives. All these facts imply the necessity to introduce process versions into rapid prototyping or, in general, into a heterogeneous processing environment to be able to exploit the properties of the different architectures in parallel and to enable an efficient mapping of the function elements to the architecture components.

We have introduced a design environment which is capable of representing, storing, and accessing the versions, thereby enabling rapid prototyping with process versions. The user predefines versions of the process structure, high–level optimizations, and function element granularity to improve the function element architecture matching and, thereby, the system performance.

References

[1] A. Balboni, W. Fornaciari, and D. Sciuto, "Cosynthesis and Co–Simulation of Control–Dominated Embedded Systems", in *Design Automation for Embedded Systems*, Vol. 1, Nr. 3, 1996.

[2] E. Barros and W. Rosenstiel, "A Clustering Approach to Support Hardware/Software Partitioning", in J. Rozenblit and K. Buchenrieder, editors, *Computer–Aided Software/Hardware Engineering*, pp. 230–262, IEEE Press, 1995.

[3] B. P. Dave and N. K. Jha, "CASPER: Concurrent Hardware–Software Co–Synthesis of Hard Real–Time Aperiodic and Periodic Specifications of Embedded System Architectures", in *Proc. of the Design, Automation and Test in Europe Conference*, pp. 118–124, 1998.

[4] E. Dirkes Lagnese and D. E. Thomas, "Architectural Partitioning for System–Level Design", in *Proc. of the 26th Design Automation Conference*, pp. 62–67, Las Vegas, USA, 1989.

[5] R. Ernst, J. Henkel, and T. Benner, "Hardware/Software Cosynthesis for Microcontrollers", in *IEEE Design & Test of Computers*, Vol. 10, Nr. 4, pp. 64–75, Dec. 1993.

[6] R. K. Gupta and G. de Micheli, "A Co–Synthesis Approach to Embedded System Design Automation", in *Design Automation for Embedded Systems*, Vol. 1, Nr. 1–2, pp. 69–120, 1996.

[7] J. Henkel and R. Ernst, "A Hardware/Software Partitioner Using a Dynamically Determined Granularity", in *Proc. of the 34th Design Automation Conference*, pp. 691–696, Anaheim, USA, 1997.

[8] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen, "Lycos: the Lyngby Co–Synthesis System", in *Design Automation for Embedded Systems*, Vol. 2, Nr. 2, pp. 195–235, 1996.

[9] R. Niemann and P. Marwedel, "Hardware/Software Partitioning Using Integer Programming", in *Proc. of the European Design & Test Conference*, pp. 473–479, Paris, France, 1996.

[10] A. Österling and R. Ernst, *A Formal Notation for Representing Process Versions*, Technical Report RP-98-1, Institut für Datenverarbeitungsanlagen, Technical University of Braunschweig, 1998.

[11] Stanford SUIF Compiler Group, *The SUIF Documentation*, Stanford University, Ca., USA, 1994. See http://suif.stanford.edu/suif.

[12] F. Vahid and D. D. Gajski, "Specification Partitioning for System Design", in *Proc. of the 29th Design Automation Conference*, pp. 219–224, Anaheim, USA, 1992.

[13] J. Wilberg and R. Camposano, "VLIW Processor Codesign for Video Processing", in *Design Automation for Embedded Systems*, Vol. 2, Nr. 1, pp. 79–119, 1996.