Systematic Power-Performance Trade-off in MPEG-4 by Means of Selective Function Inlining Steered by Address Optimisation Opportunities

Martin Palkovic Miguel Miranda Francky Catthoor‡
IMEC, Kapeldreef 75, 3001 Leuven, Belgium.
‡ Also Professor at Katholieke Univ. Leuven.
{palkovic,miranda,catthoor}@imec.be

Abstract

The hierarchical structure of real-life data dominated applications limits the exploration space for high level optimisations. This limitation is often overcome by function inlining. However, it increases the basic block code size, which causes a significant growth of instruction cache misses and thus performance slow-down. This effect has been confirmed on experiments with our applications.

We have developed a novel methodology for selective function inlining steered by cost/gain balance to trade-off power and performance. Although this results in a speed up, the increase of the instruction cache misses is still present, i.e. the memory power consumption is higher. This implies the possibility of the Pareto-optimal trade-offs between memory power and performance. Our methodology is demonstrated on an MPEG-4 video decoder.

1. Introduction

Traditionally, function inlining has been used to reduce computational complexity of programs by removing the overhead related to function calls [8, 21]. Also less overhead in stack saving when going from the calling to the called function is achieved. The elimination of the function call overhead in the critical path can eventually speed-up the application. The programming community has been aware of this advantages for a long time and high-level languages provide pragmas to implement this.

Recently, after the introduction of novel high-level optimisation methodologies for data dominated applications [5, 12, 11], function inlining has been proposed to increase exploration space for this techniques [1]. In this context, new inlining techniques need to be developed. Nowadays, besides declaration based and call based inlining [16], also a novel call instance based inlining approach [7, 3, 15] is used for the systematic exploration.

Function inlining increases basic block sizes [17], thus a potential increase of the instruction cache (I-cache) miss rate is also possible. Until now, the embedded compiler community has concentrated mainly on the increase of the overall code size of the application aiming on controlling the inlining side effects (see inside [17]). However the effect in the cache related aspects has been ignored, which could lead to actual slow-down instead of the speed-up.

For data dominated applications, we have developed a methodology for selective function inlining steered by (address) optimisation opportunities. This technique allows to control the effects of increasing the basic block size in instruction cache misses and thus to avoid potential slowdowns due to an excess of these. Moreover, this approach results in a systematic Pareto-optimal power-performance trade-off, which is the main contribution of this paper.

To test our approach, we have selected as application driver an MPEG-4 video decoder [18, 4]. The driver has already been optimised for memory usage [9] and after for address arithmetic [19] but only inside relevant functions (locally). Since this application has a deep function hierarchy, a lot of remaining optimisation opportunities across functional boundaries is present. Hence, it is well suited for the proposed methodology.

The paper is structured as follows. Section 2 describes our test bench and analyses the application driver. Section 3 shows that focusing only on potential gains for the inlining can create new bottlenecks in the application. To avoid this, in Section 4 we propose a novel methodology for selective inlining that has speeded up our application. This methodology enables to trade off speed vs. memory power as presented in Section 5. In Section 6 the results obtained on the MPEG-4 video decoder are discussed.

2. Application driver analysis

Our test bench is based on three video sequences of varying motion compensation complexity from low to high: Mother and Daughter (*m&d*) is a typical sequence with little

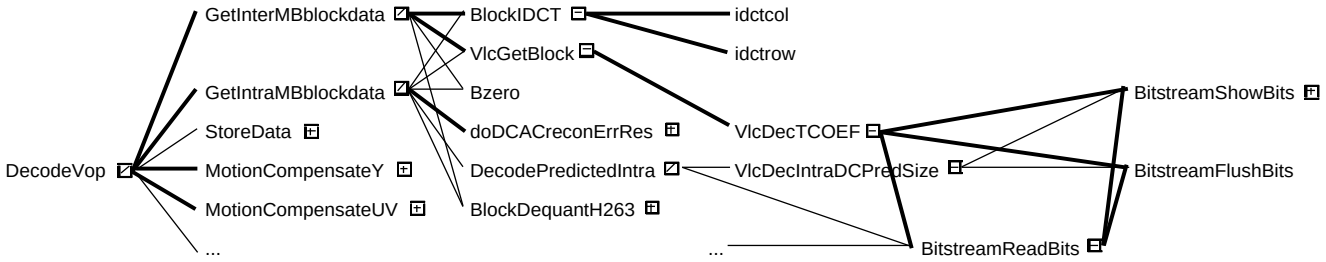

Figure 1. Call graph for the critical functions of the MPEG-4 video decoder

motion, Foreman (*for*) is a sequence with medium amount of motion and Calendar and Mobile (*c&m*) is a highly complex sequence with multiple, heterogeneous motion. The results are for the Common Interchange Format (CIF) (358 x 288) image size.

Before applying our methodology, we have profiled the application using Rational Quantify [13] and identified the core of our application as well as relevant descendants and instance tree structure (see Figure 1). Thick lines show critical path for execution time and thus important tracks for potential address optimisation.

As a characteristic feature besides its modularity, the MPEG-4 video decoder code uses generic functions instantiated with different arguments. A typical example are loops passing its iterator as an argument to the called functions. This iterator is then used in the address expressions of the called functions. Since the complex addressing is one of the main bottlenecks of our application, the nature of this data dependency for the addressing limits the optimisation opportunities and hence the overall system performance. To optimise the addressing globally, function inlining has to be used.

Function inlining can be classified into three categories (in the order of increasing flexibility) — definition based, call-based and call-instance-based. In the definition-based approach all calls to a particular function are replaced with its body. The definition-based approach, however, does not allow the freedom to specify exactly which particular calls to a function are most essential for inlining. In the call-based approach the decision to inline (or not) may be made at each call-site [6]. The call-instance-based inlining is the most flexible of the three policies where the call-graph of program is expanded to what we term as call-instance-tree. Every call is now viewed as generating a copy (or instance) of the called function [7].

Because of the very large cost sensitivity of embedded programs, we want to use call-instance-based approach to implement our inlining step. Thus our function is extended in two dimensions. The first one is the function name and the second one is the point, where the inlining should happen. This would not be possible without the systematic call-instance-based inlining approach presented in [1], which is however focused on the DTSE [5] effects and not on the address arithmetic optimisations focused in this paper.

3. Greedy function inlining

As a first attempt, we have tried to speed up our application by inlining all function instances that could have a gain while increasing the search space for address arithmetic optimisations. We refer to this approach as "greedy" because it is only steered by potential gains without taking into account associated costs.

We have inlined three major functions of our core and their relevant descendants. Thus we have aimed at enabling opportunities for further optimisations by applying the ADOPT [12, 11] address transformations after the "greedy" inlining. However, this approach has resulted in a slow-down instead of speed-up even after the application of the aggressive address optimisation techniques.

Pentium III			
#frames: 300	medium complexity		
	DTSE	ADOPT	**greedy_inl**
Decoding time [s]	6.0146	4.1560	4.3022
Framerate [fps = #frames/s]	49.88	72.18	69.73
L1-I misses [$\times 10^3$ cycles]	58308	49284	254712
#cycles I-fetch pipe stalls	210883	174701	1052541
Trimedia TM1000			
#frames: 300	medium complexity		
	DTSE	ADOPT	**greedy_inl**
Decoding time [s]	37.9259	22.7537	32.1709
Framerate [fps = #frames/s]	7.91	13.18	9.33
L1-I misses [$\times 10^3$ cycles]	99071	61447	334288

Table 1. Greedy inlining approach causes bottleneck due to stalls in L1 I-cache

The performance decrease is caused by a bottleneck in the misses of the L1 I-cache. We came to this conclusion by monitoring the architecture using the existing hardware performance-monitoring counters [14, 2] in the case of Pentium III platform, or by simulations in the case of the TriMedia TM1000. The L1 I-cache misses have grown up to a factor of 5 (see Table 1). Data misses have also increased

but much less than the instruction misses and are therefore not considered in our analysis.

For Pentium III, the instruction misses cause the instruction fetch pipe to be stalled (see Table 1). The 15% decrease of instruction misses is followed by a 17% decrease of stalls if we compare the `DTSE` (code optimised by Data Transfer and Storage Exploration [9]) and the `ADOPT` (code optimised by ADdress OPTimisation [19]) version. Moreover, the 417% increase of instruction misses causes a 503% increase of stalls comparing the `ADOPT` and the `greedy_inl` version. Thus, we can conclude that for the Pentium III instruction misses are highly correlated with the cycles where the processor gets stalled during the instruction fetch stage. One instruction miss causes approximately four cycles where the I-fetch pipe is stalled.

Unfortunately, for the TriMedia TM1000, the simulator does not provide information about the number of cycles where the I-fetch pipe is stalled. However, we expect the same correlation for this type of processor as well. This is confirmed by an increase in the number of execution cycles despite the gain due to the optimisations in the address arithmetic (see Table 1). In other words, the penalty of the greedy inlining approach has exceeded the gain due to the addressing optimisations stage.

Note that for TriMedia TM1000, the relative slow-down of `greedy_inl` version is worse than in Pentium III. Possibly, because the Pentium III architecture is highly pipelined and therefore it has a queue storing several instructions that have been already fetched from memory. Therefore, in the case of an instruction miss, the processor can still execute operations without leading necessary to stalls. This is however not the case for the VLIW architecture type like TriMedia TM1000.

The greedy inlining proves that focusing only on one parameter (such as opportunities for address optimisation) without taking into account the associated cost (such as increase in instruction misses) can lead to serious implementation bottlenecks. Section 4 describes how, by using a more selective inlining approach, we can not only avoid these bottlenecks but also further speed up the application even to trade-off speed and memory related power.

4. Selective function inlining

Our selective inlining approach related to address optimisations is formalised in terms of a cost and a gain function. The cost function reflects the instruction cycle overhead due to the increase of the biggest basic block size (after function inlining) over the capacity limit of the instruction cache set. The cost function has been chosen due to the tight relation between basic block size, L1 I-cache size and the presence of I-misses. It is important to note, that our cost function is only a first approximation, since it does not consider block boundary effects or the effect of flushing other basic blocks in the cache that could be needed later on. We believe, this approximation is sufficient for the coarse relative comparison that we need. The gain function is the opportunity for address optimisations (i.e., the amount of optimisable address related operations present in the inlined function) expressed as the number of instruction cycles gained by applying the ADOPT related transformations. The inlining exploration stage is platform dependent, because different processors may have different memory hierarchy architectures and hence, different cost parameters as well. In order to limit the search space while still tackling the real bottlenecks, we have focused on the critical path of our core.

	Cost [#cycles]	Gain [#cycles]
"if" branch	0	355
"else" branch	1640	1410

Table 2. Selective inlining exploration allows right inlining decisions

We can formalise the proposed selective function inlining approach in four steps. The first three steps aim on the selection criteria for inlining, the last step considers sorting heuristic for the selected functions list:

1. Identification of the critical functions. The goal of this step is to obtain a list of the most time consuming (relevant) functions of our application kernel. The list is sorted out according to the function relevance in the critical path. A large part of the functionality is thus "pruned" already.

FOREACH function in the list of relevant functions DO

2. Compute the cost of the function inlining. IF the biggest basic block after inlining does not fit into an instruction cache set **THEN** the number of cache lines assigned to the difference between the basic block and size of the instruction cache set is penalised (multiplied) by the average number of cycles the instruction fetch pipe stalls per miss. The obtained number of instruction cycles is the cost of the function inlining. **ELSE** Cost = 0. **END IF**

3. Compute the gain of the function inlining. The gain is the number of cycles saved due to the address related optimisations in the inlined function. This value is computed as the difference between address related operations (taking into account also the execution rate of these) before and after inlining multiplied by the average number of cycles per instruction. This average is application domain and instruction set dependent [22].

IF the cost outweighs the gain THEN this function is not suited for the inlining and thus it is deleted from the list. **ELSE** the function is kept in the list. **END IF**

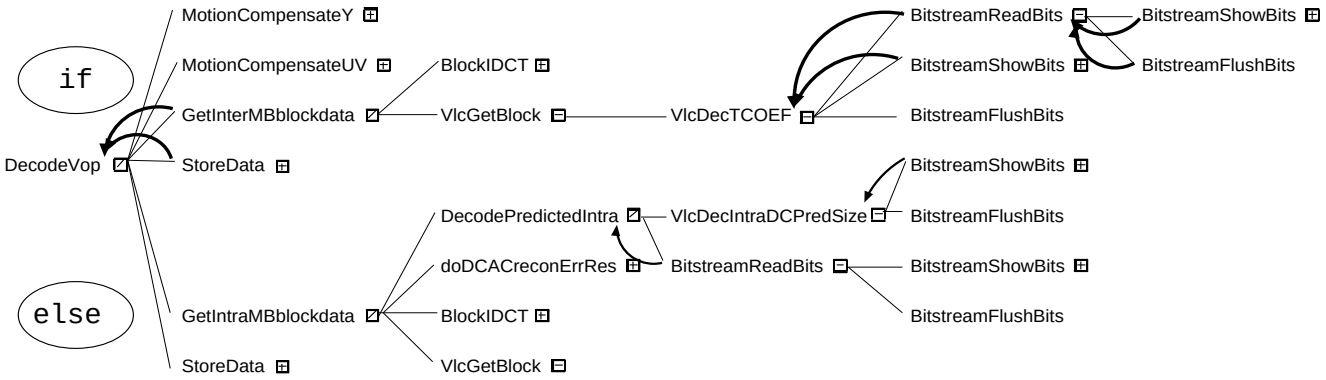

Figure 2. Call instance tree for the critical part of the MPEG-4 video decoder core

END FOREACH

4. Sort the functions suited for inlining and decide which functions to inline. The functions suited for inlining have to be sorted due to max/min analysis (maximal reduction of arithmetic related operations and minimal increase of the basic block size). Different amounts of selective function inlining lead to trade-off between speed and power. The outcome is a Pareto trade-off curve that only contains interesting design points.

In the MPEG-4 video decoder kernel, two critical parts exist, namely the "if" branch (consuming 65.48% of execution time) and the "else" branch (consuming another 13.92%). From the proportion of execution time consumption, we can conclude a priori that the "if" branch is a more promising choice for exploring inlining alternatives then the "else" branch. Using a selective inlining approach, we can decide which functions are best suited to inline due to the cost and gain balance. Table 2 shows the selective inlining exploration of the "if" and "else" branch. The numbers indicate cost as instruction cycle penalty due to stalls in the instruction fetch pipe and the gain in terms of gained instruction cycles thanks to the address related transformations.

By applying our methodology, we have decided to mostly inline in the "if" branch and its function descendants to benefit of the largest gain. In the "else" branch we will still inline some small functions at the bottom of the function hierarchy since for those is the cost/gain still beneficial. These function inlining candidates are selected following the same approach described above.

Figure 2 depicts the function inlining decisions that would lead to gains in the cost-opportunity balance in the call instance tree of our core (the combined motion core presents functionality for decoding VOPs). Lines with arrows show the different inlining possibilities. The thickness of the lines with arrows shows different degree of inlining (light inlining and heavy inlining).

5. Power vs. performance Pareto exploration

Function inlining removes function call overhead and enables opportunities for global optimisation, however, it increases the instruction transfer and storage ("memory") related cost. This performance(speed)-cost(power) trade-off leads to the need of identifying Pareto curves to achieve a good balance between these conflicting issues in the application. A Pareto curve [20, 10] is a powerful instrument that allows to make the right trade-offs at the system level. It only represents the potentially interesting points in a search space with multiple axes and it excludes all the solutions which have an equally good or worse solution for all the axes.

The memory cost increases when decreasing the execution time (e.g., due to the bigger amount in closely related instructions to be loaded in the I-cache when inlining the code) and this causes a larger energy consumption. However in this case the lower execution time is not obtained by reducing the number of memory accesses as in the case of memory oriented optimisations [5, 1] but by a larger reduction of address expressions complexity.

The mentioned trade-offs have their range of applicability before it becomes an overhead as motivated in Section 3, where the amount of function inlining has lead to instruction transfer saturation. The most interesting range of a Pareto graph is however for cases where this saturation does not happen yet. To achieve that, we have shown in Section 4 how to select the appropriate function instances candidates to have a positive balance between cost and gain. After analysing the cost-gain issues, we can tune the amount of inlining in a controlled manner and obtain different Pareto points. The different Pareto points are represented by different inlining decisions.

The input of our systematic Pareto exploration methodology is a list of functions suited for inlining (see Section 4). This list is sorted according to max/min analysis when inlining the selected function. By considering more

and more functions from the list for inlining, we obtain Pareto-optimal points. This way we create a range of solutions from minimal power consumption and minimal speed-up till maximal power and maximal speed-up.

In our exploration, we have considered two possibilities as Pareto points representatives, namely light inlining (thick arrow lines) and heavy inlining (all arrow lines) as depicted in Figure 2. The results obtained for this two cases are discussed in Section 6. However, obtaining more detailed Pareto curve is possible by inlining less functions at once.

6. Results

To compare the greedy function inlining approach (see Section 3) and our novel selective function inlining approach (see Section 4) we have run the `greedy_inl` and the best `select_inl` version on Pentium III and TriMedia TM1000 architectures. The achieved speed-up between this two approaches is more significant in TriMedia TM1000 30%–50% compared to Pentium III processor 10%– 20% (see Table 3). The significant speed-up in TriMedia TM1000 is not due to the better performed exploration in this architecture as we will discuss later, but due to a larger overhead caused by the greedy function inlining approach.

Pentium III			
#frames: 300	low complexity		
	greedy_inl	**heavy_inl**	**Δ [%]**
Decoding time [s]	1.6380	1.4961	9.48
Framerate [fps = #frames/s]	183.15	200.52	9.48
#frames: 300	medium complexity		
	greedy_inl	**heavy_inl**	**Δ [%]**
Decoding time [s]	4.3022	3.8195	12.64
Framerate [fps = #frames/s]	69.73	78.54	12.64
#frames: 150	high complexity		
	greedy_inl	**heavy_inl**	**Δ [%]**
Decoding time [s]	4.6368	3.9086	18.63
Framerate [fps = #frames/s]	32.35	38.38	18.63
Trimedia TM1000			
#frames: 300	low complexity		
	greedy_inl	**light_inl**	**Δ [%]**
Decoding time [s]	12.8384	8.7059	47.47
Framerate [fps = #frames/s]	23.37	34.46	47.47
#frames: 300	medium complexity		
	greedy_inl	**light_inl**	**Δ [%]**
Decoding time [s]	32.1709	22.0876	45.65
Framerate [fps = #frames/s]	9.33	13.58	45.65
#frames: 150	high complexity		
	greedy_inl	**light_inl**	**Δ [%]**
Decoding time [s]	28.7755	22.4435	28.21
Framerate [fps = #frames/s]	5.21	6.68	28.21

Table 3. Greedy vs. selective inlining

Our function inlining exploration has been mainly tuned to the Pentium III architecture by taken into account the parameters of that one rather than the ones corresponding to TriMedia (i.e., the instruction cache parameters, measured basic block code sizes at the assembly level, the profiling of the performance, etc.). Also the addressing oriented transformations performed earlier [19] have been mainly focused in that architecture.

For the TriMedia TM1000 we have basically run the resulting code versions but without actual systematic tuning to its architecture. Hence, resulting in a sub-optimal mapping. Still the greedy inlining approach has resulted in overhead also for this architecture (see Table 3) and our selective inlining approach has not slowed down but speeded up the application (see Table 4). However, the gain is very small since the exploration has not been tuned to this architecture.

Tuning the amount of inlining for our two target architectures results into a search space of optimal Pareto points (see Section 5). For the Pentium III, three points representing the `ADOPT` version, the `light_inl` and the `heavy_inl` version have been obtained (see Figure 3). For the TriMedia TM1000 only two points have been explored (see Table 4), since the third point led to an abrupt increase of power with small gain in execution time which falls within the error margin of the TriMedia TM1000 simulator accuracy.

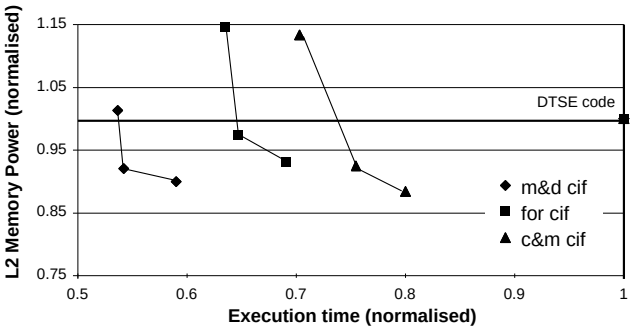

Figure 3. Pareto curve for Pentium III

From Table 4 we can conclude that for the Pentium III, in order to achieve an extra 10% in performance it is necessary to sacrifice about 20% in power. In the case of the Pentium III architecture the power corresponds mostly to the one consumed by the unified L2-cache. Here we consider misses coming from both L1 instruction and L1 data cache. We have measured the misses in the L2-cache and we have seen that the amount of these misses is negligible. In the case of the TriMedia TM1000 architecture this is the one consumed in the off-chip SDRAM.

Not surprisingly, for the TriMedia TM1000 architecture the range in performance gain achieved for a similar power increase is much more limited. This clearly indicates the need for a more architecture specific inlining exploration phase.

	Pentium III					Trimedia TM1000			
#frames: 300	low complexity								
	DTSE	**ADOPT**	light_inl	**heavy_inl**	**Δ [%]**	DTSE	**ADOPT**	**light_inl**	**Δ [%]**
Decoding time [s]	2.7880	1.6453	1.5111	1.4961	9.97	17.6723	8.9999	8.7059	3.38
Framerate [fps = #frames/s]	107.60	182.34	198.53	200.52	9.97	16.98	33.33	34.46	3.38
L1 (I+D) misses [$\times 10^3$ cycles]	255384	229788	235092	258900	12.67	132819	78818	101786	29.14
#frames: 300	medium complexity								
	DTSE	**ADOPT**	light_inl	**heavy_inl**	**Δ [%]**	DTSE	**ADOPT**	**light_inl**	**Δ [%]**
Decoding time [s]	6.0146	4.1560	3.8896	3.8195	8.81	37.9259	22.7537	22.0876	3.02
Framerate [fps = #frames/s]	49.88	72.18	77.13	78.54	8.81	7.91	13.18	13.58	3.02
L1 (I+D) misses [$\times 10^3$ cycles]	386520	359712	376224	443220	23.22	296790	174681	229331	31.29
#frames:150	high complexity								
	DTSE	**ADOPT**	light_inl	**heavy_inl**	**Δ [%]**	DTSE	**ADOPT**	**light_inl**	**Δ [%]**
Decoding time [s]	5.5580	4.4474	4.1950	3.9086	13.78	30.9765	23.1603	22.4435	3.19
Framerate [fps = #frames/s]	26.99	33.73	35.76	38.38	13.78	4.84	6.48	6.68	3.19
L1 (I+D) misses [$\times 10^3$ cycles]	254430	225048	235332	288528	28.21	195716	97755	151673	55.16

Table 4. Speed & power exploration using a selective inlining methodology

7. Conclusions

In this paper we have introduced a novel methodology in data dominated applications for systematically trading-off power and performance by using a selective inlining approach steered by address optimisation opportunities. Without this approach, the negative effects of inlining overweight can result in a significant slow-down of the application. Moreover, by using selective inlining methodology the application can be speeded up further. The results have been demonstrated on an MPEG-4 video decoder using standard processors with multimedia extended features.

8. Acknowledgements

We thank Kristof Denolf and Peter Vos from the MICS group at IMEC for providing us with the memory optimised version of the MPEG-4 video decoder and for useful feedback and comments to this paper.

References

[1] M. J. Absar, F. Catthoor, and K. Das. Function Inlining to Increase Data Access Related Optimization Opportunities. Technical report, IMEC, 2001.

[2] G. Ammons, T. Ball, and J. T. Larus. Exploiting Hardware performance counters with flow and context sensitive profiling. In *ACM SIGPLAN Conference on Programming Language, Design and Implementation*, 1997.

[3] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A Comparative Study of Static and Profile-Based Heuristic for Inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, January 2000.

[4] S. Bauer et al. The MPEG-4 Multimedia Coding Standard: Algorithms, Architectures and Applications. *Journal of VLSI Signal Processing*, 23(1):7–26, October 1999.

[5] F. Catthoor, K. Danckaert, S. Wuytack, and N. Dutt. Code transformations for data transfer and storage exploration preprocessing in multimedia processors. *IEEE Design and Test of Computers*, 18(2):70–82, June 2001.

[6] P. P. Chang, S. A. Mahlke, and W. Y. Chen. Profile-guided Automatic Inline Expansion for C Programs. *Software-Practice and Experience*, 22(5):349–369, 1992.

[7] K. D. Cooper, M. W. Hall, and K. Kennedy. A Methodology for Procedure Cloning. *Computer Languages*, pages 146–160, 1989.

[8] J. W. Davidson and A. M. Holler. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transaction on Software Engineering*, 18(2), 1992.

[9] K. Denolf, P. Vos, J. Bormans, and I. Bolsens. Cost-efficient C-Level Design of an MPEG-4 Video Decoder. In *Workshop on Power and Timing Modeling, Optimization and Simulation*, Goettingen, Germany, 13-15 September 2000.

[10] H. Eschenauer et al. *Multicriteria Design Optimization: Procedures and Applications*. Berlin, Springer-Verlag, 1990.

[11] C. Ghez, M. Miranda, A. Vandecappelle, F. Catthoor, and D. Verkest. Systematic high-level Address Code Transformations for Piecewise Linear Indexing: Illustration on a Medical Imaging Algorithm. In *Proc. IEEE Workshop on Signal Processing Systems (SIPS2000)*, Lousiana, USA, 2000.

[12] S. Gupta, M. Miranda, F. Catthoor, and R. Gupta. Analysis of high-level address code transformations for programmable processors. In *Proc. ACM Design Automation & Test in Europe Conf. (DATE)*, Paris, March 2000.

[13] http://www.rational.com/products/vis_quantify/index.jtmpl.

[14] http://www.scl.ameslab.gov/Projects/Rabbit/index.html.

[15] S. Jagannathan and A. Wright. Flow-Directed Inlining. In *ACM SIGPLAN Conference on Programming Language, Design and Implementation*, pages 193–205, 1996.

[16] O. Kaser and C. R. Ramakrishnan. Evaluating Inlining Techniques. *Computer Languages*, 24:55–72, 1998.

[17] R. Leupers and P. Marwedel. Function inlining under code size constraints for embedded processors. In *Proc. Intnl. Conf. on Computer Aided Design (ICCAD)*, pages 253–259, 1999.

[18] MPEG subgroups. MPEG-4 Overview — (V.18 — Singapore Version). ISO/IEC JTC1/SC29/WG11 N4030, Singapore, March 2001. http://www.cselt.it/mpeg/standards/mpeg-4/mpeg-4.htm.

[19] M. Palkovic, M. Miranda, K. Denolf, P. Vos, and F. Catthoor. Systematic Address and Control Code Transformations for Performance Optimisation of a MPEG-4 Video Decoder. In *15th Intnl. Conference on VLSI Design*, Bangalore, India, 7-11 January 2002.

[20] V. Pareto. *Cours D'Economie Politique*, volume I–II. Lausanne, 1896.

[21] R. W. Schiefler. An Analysis of Inline Substitution for a Structured Programming Language. *Communications of the ACM*, 20(9):647–654, 1977.

[22] D. Talla and L. K. John. Execution Characteristics of Multimedia Applications on a Pentium II Processor. In *Proc. of 19th IEEE Intnl. Performance, Computing, and Communications Conference (IPCCC)*, pages 516–524, 2000.