

Impact of Shared Instruction Memory on Performance of FPGA-based MP-SoC Video Encoder

Ari Kulmala, Erno Salminen, Olli Lehtoranta, Timo D. Hämäläinen, and Marko Hännikäinen
Tampere University of Technology, Institute of Digital and Computer Systems,
P.O. Box 553, Korkeakoulunkatu 1, FI-33101 Tampere, Finland
ari.kulmala@tut.fi

Abstract—The impact of shared instruction memory on performance is measured and analyzed for an FPGA-based Multiprocessor System-on-Chip (MP-SoC) with an MPEG-4 video encoding application. Our MP-SoC architecture allows arbitrary scaling of the number of synthesized processors and includes a monitoring unit for memory transfers. Based on the measurements with up to four processors on Altera Stratix 1S40, an estimate of the effect of the shared memory for larger configurations is presented. The shared instruction memory is shown to be area-efficient and sufficient in performance for configurations up to five processors, as the drop in encoded video frame rate stays below one compared to distributed instruction memory organization.

I. INTRODUCTION

Current FPGAs allow true multiprocessor system implementations with synthesized soft processor cores. However, FPGAs still have very limited on-chip memory, which becomes the restricting factor long before the logic resources of the chip. Therefore, it is essential to have also external off-chip memories, but those are slower and take lots of I/O pins especially when multiple on-chip processors should have separate memories. Typically, on-chip memories are used as a cache memories and a single external memory would save space.

Our case study is a scalable MPEG-4 Simple Profile video encoder that follows master-slave configuration and Single Program Multiple Data (SPMD) parallel processing paradigm. The encoder is software based and can be ported from ANSI-C source code to any processor model. This allows very rapid implementation on different multiprocessor platforms.

This work is based on our single-chip FPGA Multiprocessor System-on-Chip (MP-SoC) platform, which makes experimentation and exploration of different processors, memories and interconnection choices convenient. A custom hardware monitor is designed and integrated to precisely track the memory

activity straight from the pins. A system with one, two, and three encoding slave processors and one master processor is measured and based on the results, an estimation to evaluate the performance with larger configurations is presented.

This paper focuses on the instruction memory, which is a critical issue for scalability of performance as the number of processors is increased. Instruction memory is accessed every clock cycle, which prefers local caches. The efficiency of a cache depends on many factors, but the most significant is size. The results in [1] show that misses due to cache size are constantly over 60% of the whole amount of misses. A good compromise between limited resources and performance is the cache size of 8 KB. The details of cache design and performance are not considered in this paper.

This paper is organized as follows. First, we review the related work in Section II. In Section III, the MP-SoC architecture is described and in Section IV, the application is briefly explained. The measurements are explained in more detail in Section V. Results are presented in Section VI. Finally, in Section VII we draw the conclusions.

II. RELATED WORK

The research on shared memory multiprocessors has concentrated mainly on shared memory systems where both data and instructions are in the same memory. In [2], a cache-coherent shared memory multiprocessor is modeled and the memory subsystem has been found to be the primary system bottleneck. System performance degrades drastically when the number of processors increases, with cache miss ratio being under 2%. With four processors in [3], the shared memory (and network) contention accounted for 1.3% to 17% of the total application execution time.

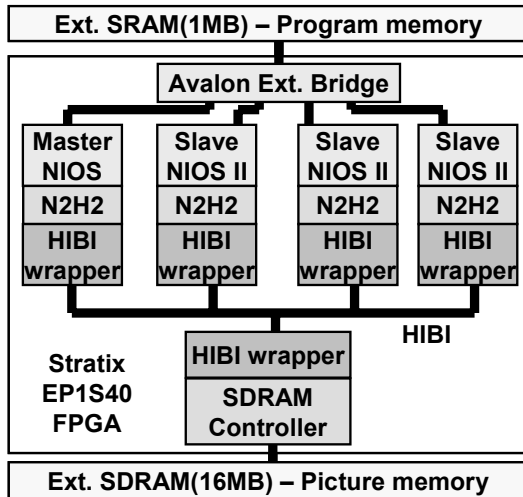


Fig. 1. Architecture of the MP-SoC.

Specifically for FPGAs, some work is presented in [4], where a multiprocessor system utilizing Nios processors for parallel LU optimization has been done. Shared instruction memory is used with a special prefetch unit that feeds the caches of the processors to reduce the shared instruction memory contention. They have achieved parallel efficiency of up to 98%, 94% and 92% with two, three, and four CPUs, respectively. However, the effect of the shared instruction memory alone is not considered, but they have noticed that cache configuration can make a difference of 20% on the performance.

For a parallel application utilizing shared data memory, the parallel efficiencies achieved with multiple processors can be found in many publications. For FPGAs, James-Roxby *et al.* [5] have inspected JPEG2000 parallelism and used a shared data memory and SPMD program style for Xilinx FPGA and Microblaze CPUs. For 2D DWT, the data accesses overwhelm the bus with four processors and no benefit can be gained by adding more processors. Before that, parallel efficiencies of around 90%, 86% and 70% for two, three and four processors have been gained.

In contrast, we concentrated in the effect of a shared instruction memory with respect to its impact on the system performance. Especially, the absolute values of memory contention were measured. Two main points were analyzed: how does the instruction memory sharing alone affect the performance and how long a simple one-level cache memory hierarchy is a sufficient option for the instruction memory sharing.

III. FPGA BASED MP-SOC ARCHITECTURE

The main objectives of our MP-SoC architecture development were to provide a flexible, yet highly

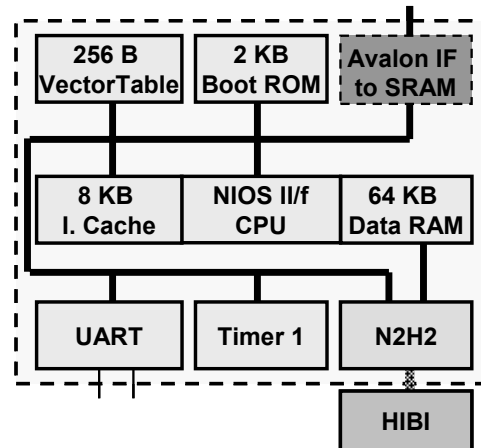


Fig. 2. The block diagram of the slave processor.

scalable platform for a range of applications. The architecture designed for MPEG-4 encoder is shown in Fig. 1. The architecture is composed of Intellectual Property (IP) blocks, which are soft-core processors (Nios and Nios II), custom on-chip network Heterogeneous IP Block Interconnection v.2 (HIBI), a Direct Memory Access (DMA) unit Nios-to-HIBI v.2 (N2H2) and custom SDRAM controller.

The used FPGA device is Altera Stratix 1S40 embedded in Nios development board Stratix professional edition. The board includes 1 MB SRAM and 16 MB SDRAM memories as well as IO devices like Ethernet. The FPGA is occupied by one 32-bit Nios processor [6] and a maximum of three Nios II processors [7]. Nios II processors are used to encode the video, while Nios is a master and collects the bitstream and handles data transfers back and forth a PC via Ethernet connection. Altogether, the MP-SoC implementation takes 70% out of the 41250 logic elements on the FPGA.

The block diagram of the slave processor is depicted in Fig. 2. Nios II core is configured as fast (Nios II/F) with 8KB of instruction cache, 2 KB of boot ROM, 256 bytes for vector table and 64 KB of data RAM. Furthermore, it contains an UART, timer and N2H2 DMA to join to HIBI. The instruction cache is direct-mapped and has eight 32-bit words in a cache line. It also utilizes dynamic branch prediction using 2-bit branch history table. The processor has two internal Altera Avalon buses [8], for instructions and data. For simplicity, only the data bus is drawn in Fig. 2. The instruction bus is connected to the vector table, boot ROM, and external SRAM. It should be noted that the external SRAM connection is essentially only an interface in slave processor. The access arbitration

logic to SRAM (Avalon ext. bridge in Fig. 1), is located within the master processor.

The master processor has the same basic configuration as the slaves, the biggest difference is the CPU type. Nios processor uses 16-bit instructions, so two instruction words fit into the 32-bit instruction bus. The cache uses branch-not-taken prediction logic. Nios processor is used as the master for a couple of reasons. First, the Ethernet stack was developed only for Nios. Second, the hardware monitor we are using needs to acquire signals from the processor core and Nios II does not allow this sufficiently due to core encryption.

HIBI is a very efficient, vendor independent on-chip multiprocessor network [9]. In this study, it is configured as 32-bit bus with round-robin arbitration and limited maximum data transfer length to avoid bus starvation. HIBI is used only for data transfers, which is out of the scope in this paper. An efficient DMA controller was designed to connect Nios and HIBI. The N2H2 unit can execute all the data transfers independently of the CPU. There is one channel for transmitting data and parameterizable number, on this occasion eight, of RX channels to receive data from different sources. N2H2 can be configured, for example, to wait for certain amount of data from the given source and store it to a memory location the CPU has specified. Multiple receiving channels are extremely useful when data comes irregularly or otherwise segmented compared to a case where every separated data block transmission causes a CPU interrupt. Thus, N2H2 frees CPU capacity for computation and handles multiple data transfers in parallel.

In this case, instructions are fetched by the Avalon bus, which is based on master and slave agents. The Avalon master can access slaves, but slaves can only request access by raising an interrupt. Masters cannot directly communicate with each other and slaves cannot communicate with other slaves. Multiple masters are supported and the access to the peripheral is arbitrated depending on the priorities of the masters. Avalon separates signal lines for reading and writing data. Address signals are shared for both read and write and are controlled by the master. If the slave is not ready, the master has to keep its signals still until the slave is ready again. The lower priority master has to wait and cannot withdraw and perform some other tasks in the meanwhile.

IV. MPEG-4 SIMPLE PROFILE APPLICATION

Our parallel video encoder software is based on horizontal spatial parallelization [10]. Briefly, a frame

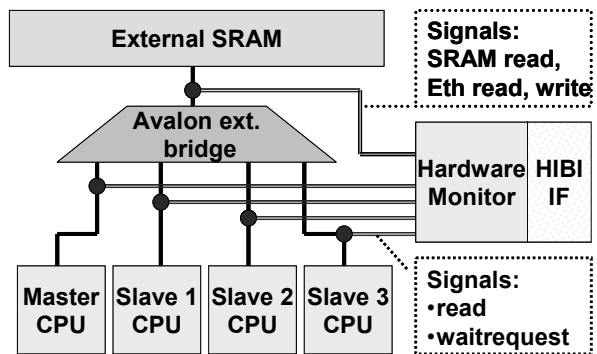


Fig. 3. Logical view of the attachment of the hardware monitor to the buses.

is horizontally divided to even sized blocks for each slave, resulting in almost equal load on them. Since the results published in [10], the MP-SoC architecture has gone through several changes, but the application has stayed the same.

The encoder is very complex and requires 140 KB instruction memory (slave processors), which is much more than available on-chip. The external 1 MB SRAM memory is split in two, a half for the master and another for the slaves. All the slaves are identical and therefore run the same program from the same location, but not in a cycle synchronized way. The execution flow is heavily dependent on the data so the execution paths differ. The master processor has instructions of its own. The video encoding program can be classified as SPMD application. However, as the master processor executes different instructions, the whole system is not SPMD.

V. MEASUREMENTS

Measurements were done with a custom hardware monitor, which was attached to the buses that go to the external SRAM as depicted in Fig. 3. The access to the shared, external SRAM is arbitrated and multiplexed by the Avalon external bridge. Note that actually both data and instruction master have to access the external SRAM. Instruction master fetches instructions, whereas data master has to access the instruction memory every now and then to retrieve compile-time constants that are stored in the code segment.

Nevertheless, read accesses to the instruction memory by both data and instruction master are accounted. No difference is made whether the access is a data or instruction fetch since effectively the result is same: the bus is reserved for certain amount of time. In the FPGA development board, SRAM bus is also shared with Ethernet. If Ethernet connection is used, also that activity can be measured. Additionally, the total amount of read accesses to the SRAM are

TABLE 1.
CLOCK CYCLES FOR CARPHONE QCIF SEQUENCE WITH THREE
ENCODING CPUS.

Monitoring Information:		Master	Slaves avg.
T_{fet}	Elapsed Time	4 567 100	
For each processor:			
T_w	Wait cycles	1 221	42 593
A_r	Read single 32b words	10 608	248 965
L	Max. latency	3	4
S	Max. block size	2	10
A_b	Separate block reads	10 605	34 653
Other:			
A_2	Two CPUs read simult.	100 819	
A_3	Three CPUs read simult.	13 487	
A_4	Four CPUs read simult.	406	
A_t	Total read amount	757 503	
A_{sp}	Longest continuous period SRAM is read	377	
A_{sb}	Number of continuous SRAM read periods	93 868	

measured from the SRAM bus to be sure that the hardware counter is operating correctly.

The measurements can be done by using a simple set of signals as shown in Fig. 3. For each processor, only read and waitrequest signals are required. Read is required for SRAM and read and write for Ethernet. However, in the benchmarked configuration, Ethernet was not utilized.

The measurements were done by encoding a regular QCIF-sequences *carphone*, *foreman*, *news*, *miss America*, and *salesman*. Each sequence was run 20 times and for 120 frames per iteration. Configurations of one, two and three encoding slaves were benchmarked. SRAM usage statistics were requested for each encoded frame and average of all frames and iterations was calculated. The hardware monitor was controlled by the master processor. The monitor was initiated just before slaves started the encoding and the results were demanded after slaves had finished the encoding. This arrangement causes some overhead because the monitor is initialized little before the encoding starts and the results are obtained a little while after the encoding has actually ended. After that, results are sent to PC via the UART connection.

VI. RESULTS

Measurement results for a 50 MHz system with one master and three slaves are shown in TABLE 1. The measured encoding speed is 10.95 frames/s for QCIF. To estimate the overhead of the hardware monitor, it was compared with the encoding speed measured by

the software, which is 10.97 frames/s. The error is only 0.2%. Probability of an instruction fetch is about 5.5% (A_r/T_{fet}) for a slave. The actual cache miss-rate cannot be monitored, because the signals between processor and cache are not accessible.

To evaluate the effect of shared instruction memory, the wait (T_w) caused by increasing the number of encoding processors was inspected. It was measured that T_w for the master is by far smaller than for slaves. As the slaves do the encoding in parallel, the total amount of wait cycles is not the total overhead, since also the waiting is done in parallel. Consequently, only the wait cycles in the critical path matter. Since the encoding load is divided quite evenly between the slaves, it can be approximated that the critical path goes through the slave that has the maximum amount of wait cycles. The maximum wait for one, two, and three slave configurations is depicted in Fig. 4. The total time elapsed for encoding a frame (T_{fet}) for the same configurations is shown in Fig. 5.

The SRAM utilization is only 16.6% (A_t/T_{fet}) with three encoding slaves. As the probability for waitrequest per each data read is 17% ($\sum T_w/A_r$) and utilization is that low, it shows that the accesses are mostly bursts of a few words for all of the processors. In addition, all four processors rarely (0.05% of all reads, A_4/A_t) try to access the memory at the same time and the simultaneous request by three processors is also quite scarce (1.8%, A_3/A_t), which implies that the memory accesses to the instruction memory interleave well.

To estimate the overhead caused by shared instruction memory, the following assumptions were made. First, the data accesses do not overwhelm HIBI. This is reasoned, since HIBI transfer bandwidth with 50 MHz is theoretically at lowest 95 MB/s and for 40 frames/s, this system transfers only about 14 MB/s. On the other hand, if each slave fetches a word with a probability of 5.5%, the external instruction memory bus saturates after 18 slaves. Furthermore, a read from the instruction memory does not necessarily stall the execution, since cache prefetching is used. The occurrence of prefetching cannot be monitored, so we assume a worst-case condition, expecting that each instruction fetch does stall the execution. Therefore, the drawback of the shared instruction memory is most probably less than expected by the figures.

In order to create the estimate, we have to extract two components of the results: the overhead caused by shared memory and the expected performance without shared memory. First we consider the caused overhead, specifically the amount of wait cycles in the critical path. The estimation is fitted to a 2nd-order

curve based on the measurements as shown Fig. 4. The number of wait cycles is calculated as

$$T_{w\max}(N_s) = 1523 \cdot (N_s)^2 + 15081 \cdot (N_s - 1) + 130, \quad (1)$$

where N_s is the number of slave processors ($N_s > 0$). The error of this estimation is under 0.1% compared to measurements. Both the estimated and measured curve are drawn in Fig. 4. The curve is measured to be strongly linear. However, we assume that as the number of processors increase, that causes a slightly more than linear overhead due to increasing activity on the shared memory lines.

Second, the measured wait cycles caused by shared memory were subtracted from the total encoding time to have an estimate of the encoder performance without shared memory overhead. Since the performance does not increase linearly as can be seen from Fig. 5, we have derived a power function to approximate it as follows.

$$T_{frame}(N_s) = (T_{fet} - T_w) \cdot (N_s)^{-0.94}, \quad (2)$$

where T_{frame} is the frame encoding time, T_{fet} is the measured encoding speed of one frame with one processor, T_w the measured wait, and N_s is the number of slave processors ($N_s > 0$). The exponent value of -0.94 is used to approximate the gain with each processor – the overall speed-up decreasing compared to linear when the number of processor increases. For example, with two slaves we have better gain per processor than with three processors. The measured and estimated T_{fet} are shown in Fig. 5. The measured values are the real values with wait cycles, including no estimations. The estimation is $T_{frame}(N_s) + T_{w\max}(N_s)$. The error between real values and estimated is less than 2%.

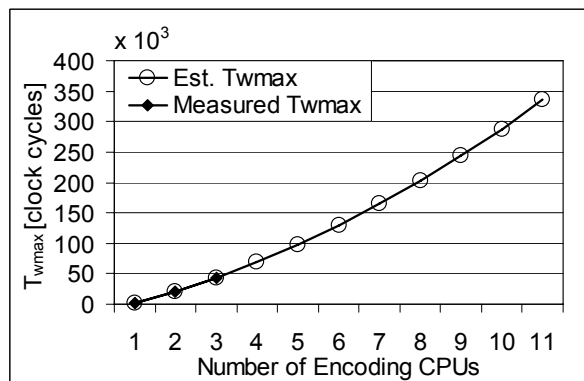


Fig. 4. Maximum number of wait cycles for one processor.

The parallel efficiency ratio is calculated as

$$Eff(N_s) = \frac{T_{frame}(N_s) + T_{w\max}(N_s)}{T_{fet} \cdot N_s} \cdot 100\%, \quad (3)$$

The measured parallel efficiencies are 97% with two slaves and 93% with three. From calculations, these are 96% for two and 94% for three. The error between measured and calculated is less than 1%.

At this point, we have derived equations for both the latency caused by the shared memory, and the performance without the wait cycles. These figures are quite accurate with a small number of processors. However, as the number of processors increase, there are certain aspects that cannot be taken into account with this kind of calculations. These include, for instance, un-equal division of macroblocks, which causes variation in performance gain.

However, our estimate with QCIF should stay quite accurate with a number of slaves CPUs range of one to eleven. After that, un-equal division of macroblocks starts to make greater difference. This is because after eleven processors, adding another CPU does not necessarily decrease the number of macroblocks to be encoded in the critical path. On the other hand, if the picture size grows from QCIF to, for example, CIF, the number of macroblocks increases and consequently the region where these approximations are accurate widens.

Finally, to conclude the effect of the shared instruction memory to the performance, we can see that the penalty is not very high as depicted in Fig. 6. The wait cycles are responsible of less than 1% of the total execution time with up to three CPUs. Furthermore, the overhead still causes less than one frame/s drop (under 4% of execution time) with five slave processors. After that, the difference starts to be quite large, with four frames/s drop with nine processors, shared memory overhead consuming about 13% of the performance, but that is still acceptable in

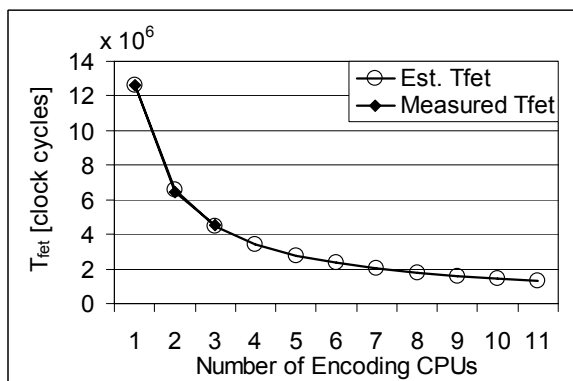


Fig. 5. Frame encoding time with shared memory overhead.

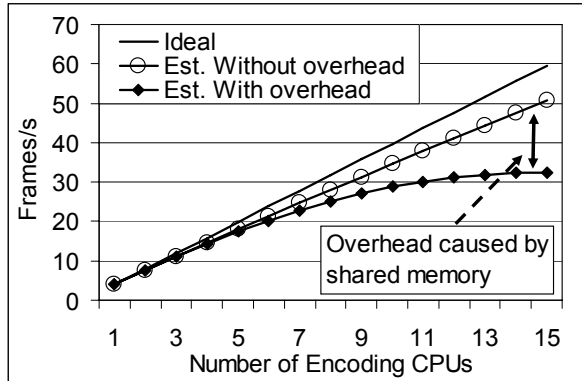


Fig. 6. Estimated frames/s increase with and without shared instruction memory overhead.

some cases. Nevertheless, these figures encourage using shared instruction memory at least with less than six processors and still with up to nine processors it is a sufficient option, even with the used one-level cache memory hierarchy. However, the larger configurations could benefit from more complex memory hierarchy.

If we consider the parallel efficiencies achieved by James-Roxby *et al.* [5], around 90%, 86% and 70% for two, three and four CPUs were achieved with shared data memory. Hence, compared to our results, we can see that the felicity of the implementation of parallelism has the greatest impact on the performance rather than the shared memory. Therefore, instruction memory sharing can be considered at least as viable option as traditional data memory sharing.

VII. CONCLUSIONS

It is very feasible to use only one instruction memory for multiprocessor systems and save in the chip costs. The shared instruction memory does not cause a drastic negative drawback with a system with less than six processors, being less than 4% of the whole execution time. The overhead caused by shared instruction memory is still adequate for systems with less than 10 processors. However, the instruction cache is compulsory, but it does not need to be state-of-the-art. A 5% probability for an instruction fetch from the shared memory suffices. As the application-specific issues, such as un-equal division of macroblocks, come to picture, the estimate no longer applies. However, at least with video encoding, that point is not reached in the range where one shared memory is feasible. In future work we are going to expand our system over to several FPGA boards and bigger FPGA chips to further evaluate the applicability of shared instruction memory.

REFERENCES

- [1] J.L. Hennessy and D.A. Patterson, "Computer Architecture – A Quantitative Approach", third edition, Morgan Kaufmann Publishers, San Francisco, USA, 2003.
- [2] R. Jog, P. L. Vitale, and J. R. Callister, "Performance evaluation of a commercial cache-coherent shared memory multiprocessor," Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM, 22-25 May 1990, pp. 173-182.
- [3] C. Natarajan, S. Sharma, and R. K. Iyer, "Measurement-based characterization of global memory and network contention, operating system and parallelisation overheads: case study on a shared-memory multiprocessor," Proceedings the 21st Annual International Symposium on Computer Architecture (ISCA), IEEE, 18-21 April 1994, pp. 71-80.
- [4] X. Wang and S.G. Ziavras, "Performance optimization of an FPGA-based configurable multiprocessor for matrix operations," Proceedings of the Field-Programmable Technology (FPT), IEEE, 15-17 December 2003, pp. 303-306.
- [5] P. James-Roxby, P. Schumacher, and C. Ross, "A single program multiple data parallel processing platform for FPGAs," Proceedings of Field-Programmable Custom Computing Machines (FCCM), IEEE, 20-23 April 2004, pp. 302-303.
- [6] Altera Corporation, "Nios 3.0 CPU Data Sheet," version 2.0., available online, March 2003.
- [7] Altera Corporation, "Nios II Processor Reference Handbook," available online, May 2005.
- [8] Altera Corporation, "Avalon Interface Specification – Reference manual," version 2.4., available online, January 2004.
- [9] E. Salminen, V. Lahtinen, T. Kangas, J. Riihimäki, K. Kuusilinna, and T. D. Hämäläinen, "HIBI v.2 Communication Network for System-on-Chip," LNCS 3133 Computer Systems: Architectures, Modeling, and Simulation. A.D. Pimentel, S. Vassiliadis, (eds.). Springer-Verlag, Berlin, Germany, July 2004, pp. 412 - 422.
- [10] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hannikainen, and T.D. Hämäläinen, "A parallel MPEG-4 encoder for FPGA based multiprocessor SoC," Proceedings of International Conference on Field Programmable Logic and Applications, IEEE, 24-26 August 2005, pp. 380 – 385.