ÆTHER: Dynamic and Self adaptive Middleware

Farooq Muhammad[1], Fabrice Muller[2], Michel Auguin[3]
University of Nice Sophia-Antipolis / I3S CNRS
2000 rte des lucioles,
Les Algorithmes - bât. Euclide B - BP 121
06903 Sophia Antipolis - Cedex, France
{muhammad[1],fmuller[2],auguin[3]}@i3s.unice.fr

Abstract— **The development of reconfigurable devices that could make themselves domain-specialized at run time is becoming more and more common. Future reconfigurable architecture will have these computing devices as basic blocks, and reconfigurable architecture could make assemblies of these devices, on the fly, to execute concurrent applications. The migration from completely generic lookup tables and highly connected routing fabrics to self adaptive specialized coarse-grain reconfigurable devices and very structured communication resources presents designers with the problem of how to best customize the system based upon anticipated usage. Then there is a need of not only exploiting parallelism from applications at micro-thread level, dynamically, but system also starves for a dynamic and self adaptive middleware to schedule these micro-threads on thousands of such computing devices. This paper focuses at the problem of dynamic allocation and scheduling of resources to numbers of applications on such architecture.**

Keywords-Resource Allocation; dynamic scheduling; self adaptive task structuring.

I. INTRODUCTION

In order to provide high performance computation power to serve the increasing need of large applications, people strive to improve a single machine's capacity or construct a distributed system composed of a scalable set of machines. Compared to the former, where the improvement is mainly up to the hardware technology development, the construction of distributed systems for resource collaboration is more complex. Some of well-known existing distributed systems composed of heterogeneous resources are Condor[4], NetSolve[5], Nimrod[6], Globus and the Grid[7] computation environment. Sabin et al [9] propose a centralized metasheduler which uses backfill to schedule parallel jobs in multiple heterogeneous sites. Similarly, Arora et al [10] present a completely decentralized, dynamic and sender-initiated scheduling and load balancing algorithm for the Grid environment. All these approaches don't deal with application model where concurrent threads are created and managed at run. These methods do not target future architecture where each resource of a processor has capability of self optimizing and interconnects with other resources to form assemblies to execute concurrent application.

ÆTHER[1] system is hierarchical both at function and architecture level. At function level, application is written which self adapts itself to well suit with the application objective and to cope with dynamic changes happening in environment. Applications are quite dynamic in nature where concurrent threads are instanced dynamically according to number of resources of system. Architecture of the system is not traditional as it is not single unit of computation. It is a network of given number of SANEs. A SANE is self adaptive networked entity which can self optimize itself.

In classical scheduling theory it has been commonly assumed that a task requires only one processor at a time for its processing. However for many practical problems this assumption is not valid. A task requires task-dependent (worst case execution time) number of processors (minimum) simultaneously for its processing.

One of the distinguishing characteristics of real-time and embedded systems is the concern over management of finite resources to guarantee requested Quality of Service (QoS) to concurrent applications. The QoS properties are the quantitative properties of the resource, such as its capacity, execution speed, reliability, and so on. In real-time and embedded systems, it is this quantifiable finiteness that must be managed. The management of resources with many applications is one of more thorny aspects of system design.

In this paper, we propose an algorithm for scheduling of hard and soft real-time tasks in an architecture which is composed of multiple processing units. These computing units are self adaptive and have the capability to optimize according to application needs. Application helps create/instantiate concurrent threads at run time. The primary goal of the proposed algorithm is to maximize the schedulability of soft tasks without jeopardizing the schedulability of hard tasks. The algorithm has the inherent feature of degrading/upgrading QoS, by dynamic managing concurrency of tasks by allocating more resources to most appropriate task i.e. (hard real time tasks are not always preferred over soft real time tasks). This algorithm helps to maximize execution of concurrent execution of tasks and distribute resources to thousands of concurrent threads of a task where objective is to ensure timeline guarantees of tasks.

A. *Definition Of The Problem*

We have n tasks $\tau = \{T_1, T_2, T_3, ..., T_n\}$ and Rx parallel resources. These parallel resources self adapt themselves according to task executing on it. These resources form assemblies and configure/self-adapt [11] connecting fabric between SANE elements to execute tasks while tasks have the capability of dynamically managing (creating threads at runtime) concurrency in it depending upon the availability of

ÆTHER is IST-FET (Information Society Technology-Future and Emerging Technologies) European project with main objective to study novel self-adaptive computing technologies for future embedded and pervasive applications.

1-4244-1553-5/07/$25.00 ©2007 IEEE

resources. Each task may have a minimum and/or maximum demand of resources to respect its deadline constraints. System resources are limited so that demands of all tasks can not be satisfied simultaneously. Execution time of task monotonically decreases with each resource allocated to it until the maximum level of parallelism in task. For some tasks where concurrent parts of tasks are dependent, execution time of a task is monotonically decreasing until a point (called threshold) and it starts increasing monotonically after this point for each allocated resource. Task T is represented as sequence of concurrent execution of µthreads (μT). Concurrency level at each sequence of task may be different and may have a value less than its minimum demand of resources, thus allocating minimum resources statically could cause system to decrease its performance.

We are interested in scheduling these tasks on Rx parallel resources such that maximum tasks could respect their deadline constraints and utilization of resources could be maximized.

The rest of the paper is organized as follows: in section 2, we define *ÆTHER* architecture. The proposed approach is presented in section 3. Section 4 presents technique to assign resources to task, followed by method description for calculating minimum resource demand of each task. Algorithm of dynamic allocation of resources is demonstrated in section 6. Schedulability analysis is illustrated in section 7. Conclusions are made in section 8.

II. ÆTHER ARCHITECTURAL MODEL

In the ÆTHER project, SANE [11] (Self Adaptive Networked Entity) is introduced as basic computing entity aims to be networked with other entities of the same type to form complete systems. Each of these entities is meant to be self-adaptive, which implies that they can change their own behavior to react to changes in their environment or to respect some given constraints.

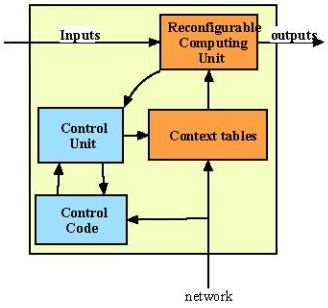

Figure 1. Self Adaptive Networked Entity (SANE)

As shown in Figure 1, the controller changes the state of the SANE hardware implementation by changing some parameters of the currently loaded task as well as changing the task to another implementation of the same task or a completely different task. The computing tasks are loaded in the computing engine. They can be described as bit-streams if the computing engine is viewed as an FPGA fabric or as binary files for a soft-processor. The existence of the monitoring process associated with an adaptation controller provides the SANE with the self-adaptation ability. The latest part of the SANE is the communication interface. It is dedicated to collaboration among the SANE hardware elements that compose the architecture. The collaboration process is done through a publish/discover mechanism that allows a SANE hardware element to publish its own abilities and parameters and to discover the computing environment formed by the other SANE hardware elements in its immediate local neighborhood. This mechanism enables the SANE hardware elements to exchange their tasks or just to clone their states to other SANE hardware elements.

The SANE processor (Figure 2) is a runtime reconfigurable architecture composed of thousands of SANE elements. These elements are interconnected dynamically and self-adaptively to execute concurrent applications. SANE processor is a multi-core processor with SANE elements as its processing cores. SANE processor has more flexibility than simple multi-core processor due to its capability of not only reconfiguring its cores (SANEs) but also restructuring the interconnection between them.

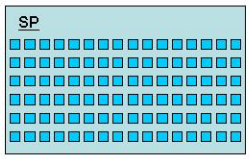

Figure 2. SANE processor composed of SANE Elements

SANE elements (equal to resources allocated to task T_i dynamically) are combined to form a SANE assembly to execute task T_i. As number of concurrent tasks at run time varies dynamically so assemblies are formed at run time. Moreover resources assigned to each task varies, hence number of SANE elements in one assembly varies dynamically as well. Number of SANE assemblies in one SANE processor changes at run time (Figure 3 (a),(b)).

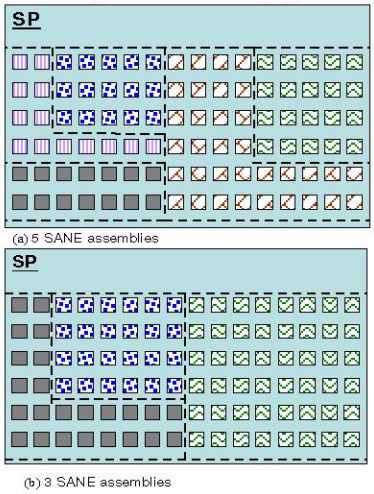

Figure 3. SANE Assemblies

III. APPLICATION MODEL

There are two types of parallelism to be exploited: task parallelism and data parallelism. In ÆTHER project, parallelism is exploited through a coordination language. S-

Net[2], used in ÆTHER project, is a coordination language that orchestrates asynchronous components that communicate with each other and their execution environment solely via typed streams.

The application program units are presented in an appropriate fully-fledged programming language, such as C, Java, etc., while the aspects of communication, concurrency and synchronization (referred to by the term coordination) are captured by a separate, coordination, language. S-Net program is compiled down to µthreadedC (μTC)[3] which is used as user programming language. μTC is a rather profound but simple extension to the C language, allowing it to capture massive thread-based concurrency. μTC is capable of expressing static heterogeneous concurrency and dynamic, homogeneous concurrency.

Only a small number of constructs are added to C, along with the semantics of the synchronizing memory. The constructs map onto low-level operations that provide the concurrency controls in a µthreaded ISA, and allow concurrent programs to be dynamically instanced and preempted, either gracefully or with a prejudice. Family identifiers provide the control over the concurrent sections.

A. Task Structure

A task T_i is represented as a sequence of concurrent execution of µthreads. Each task T_i consists of a finite series of sequences $\{T_{i,1}, T_{i,2}, ..., T_{i,j}, ..., T_{i,n}\}$. Each sequence T_{ij} consists of (max) N_{ij} concurrent μTs of execution and each μT must run for at most C_{ij} time units; such value is called the worst-case computation time of μT, S_{ij} is the slow down factor if there is dependency between two μTs.

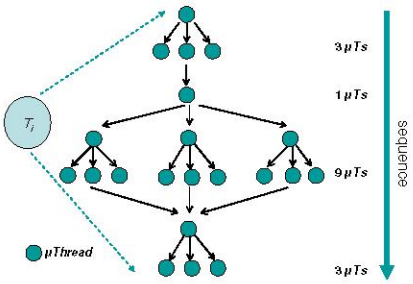

Figure 4. Representation of a task

The number of μTs in any sequence is limited by a number and is known a prior. The μTs are instanced dynamically depending upon the resources availability. In Figure 4, we can observe that in a task structure, we have different number of μTs at different levels. These μTs represents the maximum limit but threads created at runtime may have different values bounded by these maximum limits.

The resources are allocated to different tasks depending upon its criticalness, its execution time and efficiency of a resource to execute a thread. It is assumed that one resource is capable of executing a μT, but resources can optimize self adaptively to execute more than one μT at a time. In this case middleware will change the task structure dynamically and self adaptively that helps to make better decisions about resource allocation. It will increase the resource utilization in terms of minimizing wastage of resource utilization (see in section 4). Initially, execution time of a task is calculated considering that one SANE is capable of executing only one μT at a time.

1) Worst Case Execution Time:

The worst case execution time of task is function of allocated resources at run time. Worst case execution time can be calculated by considering only one μT in execution at one time *i.e* with no parallelism.

$$C_i = \sum_{j=1}^{n} N_{i,j} * C_{i,j} * S_{i,j} \quad (1)$$

2) Best Case Execution Time:

Best case execution time is calculated by assigning resources equal to are less than maximum μTs at that level

$$C_{i,b} = \sum_{j=1}^{n} S_{i,j} * \left\lceil \frac{N_{i,j}}{R_b} \right\rceil * C_{i,j} \quad (2)$$

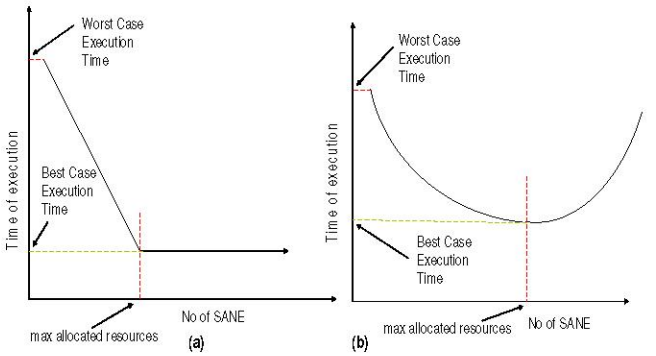

Figure 5. Execution time of task

The worst case execution time and best case execution of a function depends on number of minimum and maximum resources that can be allocated to it. If there is no dependency between μTs of one family then there time of execution will be decreased linearly. But there will be a point after which increase in allocation of resources will not further decrease time of execution. It will remain constant. This point is called saturation point. In some cases when there is strong dependency between μTs of a family then time of execution will not be decreased linearly and there will be a threshold point after which time of execution of family of μTs start increasing instead of decreasing if more resources are assigned than its threshold point.

During execution of a task, hardware resource (SANE element) self adaptively optimize its power to execute more than one μT of a family. In this scenario, task structure will be changed due to positive feedback from hardware.

B. Self Adaptive Task Structuring

SANE elements, having capabilities of self optimization, require a dynamic and self adaptive middleware to cope with these optimizations of hardware that could distribute resources in an efficient manner.

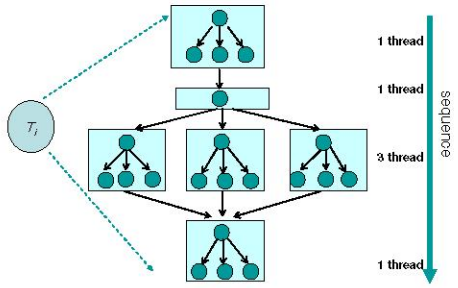

Figure 6. Varied task structure

Optimization achieved by hardware for any task can be of different types. SANE element

- could be optimized to provide dedicated functionality implemented in hardware for a task.
- could self optimize to execute complete or partial family of μTs concurrently, instead of executing one μT at a time.

1) Dedicated SANE for a Task:

If a SANE element has dedicated hardware for a task or it has self-optimized itself at runtime to execute all sequences of task, then there will be no more modifications in structure of this specific task.

2) Optimized SANE for Family of µTs of a Task:

SANE element may make itself specialized only for parts of task instead of complete task. It may provide better results for a family of µTs. In this scenario, task structure will be changed (Figure 6) and there will be need to recalculate the execution time of tasks with new parameter i.e. number of sequences, number of threads at each level (in one sequence) and worst case execution time of modified thread.

Resources are allocated to different tasks depending upon their structure and its deadline constraints.

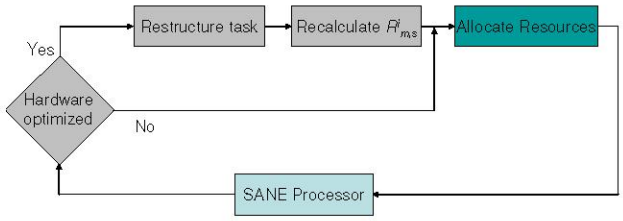

Figure 7. Restructuring of Task self adaptively

IV. RESOURCE ALLOCATION & SCHEDULING

Ideal scheduling algorithm is one where each task is assigned resources proportion to its weight during the whole length of its execution. Ideal scheduling algorithm is impractical

- in case of general purpose processors where computational capacity of one processor can't be assigned to different tasks in proportion to their weights. But in case of SANE processors, resources (SANE Elements) may be assigned proportionally. Then there is a need to have an application model which can help to execute a task in parallel to use all those resources which are allocated to it proportion to its weight.
- in real cases where application does not have μTs equal in number (or more) that corresponding to its weight. The number of μTs that an application can execute in parallel varies during its execution.

If minimum number of uthreads at any time of execution are more than number of resources corresponding to its weight then this task can be scheduled ideally.

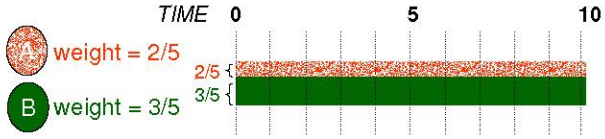

Figure 8. ideal scheduling of concurrent tasks

But if a task T_i has number of μTs less than $R^i_{m,s}$ (resources proportion to its weight) at some level and greater than $R^i_{m,s}$ at other levels (which is the case most of time), then this task can't be scheduled ideally. To provide guarantees for this task, reservation of resources more than its weight is needed. It will cause certain resources left unused and there will be wastage of resource utilization.

A. Wasted Resource Utilization

If schedulability analysis is carried out based on minimum resources $R^i_{m,s}$ then there are chances that these resources may not be fully used during execution of a task. As few sequences of a task may need less resources than $R^i_{m,s}$. It means certain percentage of resources will not be used during execution of a task.

1) Homogenous Concurrent uthreads

μTC has the capability of controlling the concurrency of μTs dynamically if μTs at that level are homogenous. In this case, if we have more than one homogenous families of μTs at same level, then these families can be considered as a single family. If two or more than two families are concurrent then number of total μTs at that level $N_{i,k}$ is sum of all those μTs.

$$N_{i,k} = N_{i,n} + N_{i,m} \tag{3}$$

Each μT of these two families will have same worst case execution time. Wasted Resource Utilization by task T_i when it is assigned $R^i_{m,s}$.

$$WR_i = \sum_{k=1}^{n} max\left(0, \left(R^i_{m,s} - N_{ik}\right) * C_{ik}\right) \tag{4}$$

where n represents number of sequential families of task T_i.

2) Heterogeneous Concurrent µthreads

If families of μTs at any level are heterogeneous i.e worst-case execution time of μT in one family is different from that of other, then calculation of wasted resource utilization may have different value than calculated in above equation and is calculated in the following way.

$$WR_i \le \sum_{k=1}^{n} max\left(0,\left(R^i_{m,s} - N_{ik}\right) * max\left(C^1_{ik}, C^2_{ik}, ..., C^m_{ik}\right)\right) \quad (5)$$

Where $(C^1_{i,k}, C^2_{i,k}, ..., C^m_{m,k})$ are worst case execution times of μTs of concurrent heterogeneous families.

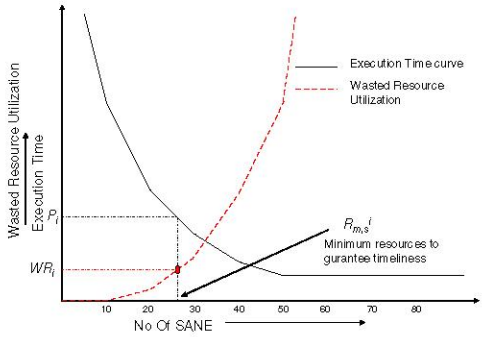

Figure 9. Wasted Resource Utilization

If Wasted Resource Utilization (WR_i) by task T_i is zero when it is assigned $R^i_{m,s}$ then

$$R^i_{m,s} = \left\lceil \frac{C_i}{P_i} \right\rceil otherwise\ R^i_{m,s} > \left\lceil \frac{C_i}{P_i} \right\rceil$$

With the new static value of $R^i_{m,s}$ task will respect its real time constraints but wastage of resources will be increased.

Wasted Resource Utilization will be Zero if

$$R^i_{m,s} \le min(N_{i1}, N_{i2}, ..., N_{in})$$

V. MINIMUM RESOURCES FOR EACH TASK

A task is represented as a sequence of concurrent executions of uthreads. At each level a task has different number of μTs and worst case execution time of a μT at one level may be different from that of a μT at other level. Moreover the slow down factor may be different at different levels.

If worst case execution time calculated for a task T_i is higher than its deadline P_i, then there is a need to recalculate minimum number of resources that a task should be assigned to guarantee its deadline constraint.

The minimum number of resources $R^i_{m,s}$ that a task should be allocated can be calculated by an iterative process. We have a function associated with each level of a task. With the help of this function we can calculate the worst case and best case execution time of this family of μTs. Calculation of minimum resources that does not correspond exactly to its weight (higher than its weight) is calculated as follows:

1. $R^i_m = \left\lceil \frac{Ci}{Pi} \right\rceil$
2. $WR_i = \sum_{k=1}^{n} max\left(0,\left(R^i_m - N_{ik}\right) * C_{ik}\right)$
3. $if\left(Ci/R^i_m + WR_i > Pi\right)$
4. $R^i_m = \frac{Ci}{Pi - WR_i}$
5. *go to step 3*
6. *else return* R^i_m

A task T_i can be allocated more than $R^i_{m,s}$ at run time. If a task T_i use more than $R^i_{m,s}$ for certain duration, then minimum resource for rest of task may have a smaller value and is calculated dynamically (dynamic minimum demand $R^i_{m,d}$).

A. Task Preemption

If a task T_i has used more than $R^i_{m,s}$, then this task T_i can be pre-empted as well. Time PT_i for which a task can be pre-empted by low priority task depends upon the time for which this task has used resources more than $R^i_{m,s}$ and how much.

$$PT_i = P_i - \left(\frac{C^r_i}{R^i_m} + WR_i\right) \quad (6)$$

A task can be pre-empted for a longer time than calculated in above equation, if it could be allocated more than $R^i_{m,s}$. It is possible in two situations:

- $\sum_{i=1}^{n} R^i_{m,s} < R_x$
- set of tasks for which $R^i_{m,s}$ and $R^i_{m,d}$ have different values and absolute deadlines of these tasks are greater than absolute deadline of T_i.

In both of these cases task T_i can be pre-empted for a longer duration. Time for which this task can be pre-empted is calculated as follows:

$$PT_i = P_i - \left(min\left(\frac{C^r_i}{R^i_m}, \frac{C^r_i}{R^i_m + \sum_{j=1}^{p} R^j_{m,s} - R^j_{m,d}} \right) + WR \right) \quad (7)$$

VI. ALGORITHM

Real-time applications are classified into two major categories of hard and soft real-time tasks (HRT and SRT tasks respectively). Hard real-time tasks have critical deadlines that are to be met in all working scenarios to avoid catastrophic consequences. In contrast, soft real-time tasks (e.g., multimedia tasks) are those whose deadlines are less critical such that missing the deadlines occasionally has minimal effect on the performance of the system.

Tasks are allocated resources depending upon its value of $R^i_{m,s}$. The calculation of minimum resources for a task changes at run time (Figure 10), if it was allocated more than $R^i_{m,s}$ during its execution.

HRT tasks can't be preempted by any task if $R^i_{m,s}$ and $R^i_{m,d}$ have same value. If there is a difference then HRT task can be preempted.

```
Algorithm: Dynamic allocation of Resources.
Whenever new family starts
        If (R_F >=Nik)              ; R_F = Free resources
            Allocate Nik;
        else
            if (R_F >=R_m^i)
                allocate R_F;
            else                                ; HRT = hard real time task
                if ( R^i_alloc(HRT) > R_m^i)    ; R^i_alloc allocated resources to Ti
                    liberate (R^i_alloc- R_m^i);
                else
                    if (low priority SRT task exist)
                        preempt lower priority Ti (SRT);
                    else
                        allocate R_F;
Whenever a family is finished (or preempted)
                Recalculate R^i_m,s;
```

Algorithm 1 Dynamic Allocation of Resources

If there is no such HRT task, where $R^i_{m,s}$ and $R^i_{m,d}$ have different values then SRT task can't preempt HRT task. In this case low priority SRT task can be preempted only by higher priority SRT task.

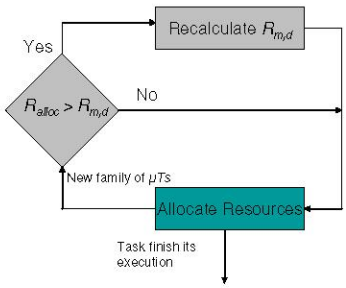

Figure 10. Recalculation of Rm

VII. SCHEDULABILITY

To get the system's behavior deterministic for HRT tasks, we must be sure that at any time minimum demand of resources for all HRT task does not exceed number of resources in architecture. Schedulability analysis for HRT tasks

$$\sum_{i=1}^{n} R^i_{m,s} \le R_x \tag{8}$$

If the sum of minimum resources is less than Rx then extra resources can be used to schedule SRT tasks. The allocation/reservation of $R^i_{m,s}$ to task T_i introduce wasted slots of resources, these wasted slots can be exploited to schedule SRT tasks. A necessary condition for scheduling of SRT tasks

$$\sum_{k=1}^{m} \frac{C_k}{P_k} \le \sum_{i=1}^{n} \frac{WR_i}{T_i + WR_i} + \left(\frac{R_x - \sum_{i=1}^{n} R^i_{m,s}}{R_x} \right) \tag{9}$$

Where n represents number of HRT tasks and there are m SRT tasks in the system.

VIII. CONCLUSIONS:

In this paper we have presented an approach for scheduling of HRT and SRT tasks where each task requires more than one resource to finish its execution. We have provided a model that allocates resources to tasks dynamically, that redefines its demand of minimum resources self-adaptively. This model restructures the task as well if SANE hardware optimizes itself. It provides a means of efficiently exploiting unprecedented computational power of SANE processor.

IX. ACKNOWLEDGMENT

This work was partially supported by the European Commission under Project ÆTHER No. FP6-2004-IST-4-027611, and by French Ministry of Higher Education and Research under contract No84-2005. http://www.aether-ist.org.

REFERENCES:

[1] Bousias, K, Hasasneh N M and Jesshope C R (2006) Instruction-level parallelism through microthreading - a scalable Approach to chip multiprocessors, Computer Journal, 49 (2), pp 211-233.

[2] A.Shafarenko (2006) The principles and construction of SNet, Internal report, Dept of Computer Science, University of Hertfordshire.

[3] Bousias, K. and Jesshope, C. R. (2005) The challenges of massive on-chip concurrency. Tenth Asia-Pacific Computer Systems Architecture Conference, Singapore, October 24-26. LNCS 3740, Springer-Verlag.

[4] Michael Litzkow, Miron Livny, and Matt Mutka, "Condor - a hunter of idle workstations," Proceedings of the 8th International Conference of Distributed Computing Systems, pp. 104-111, June 1988.

[5] Henri Casanova and Jack Dongarra, "NetSolve: a network server for solving computational science problems," The International Journal of Supercomputer Applications and High Performance Computing, vol.11, no. 3, pp. 212-223, Fall 1997.

[6] Abramson D., Sosic R., Giddy J., and Hall B., "Nimrod: a tool for performing parametised simulations using distributed workstations," The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, Aug. 1995.

[7] Ian Foster and Carl Kesselman, The Grid: Blueprint for a New Computing Infrastructure. ISBN 1-55860-475-8, July 1998.

[8] http://www.aether-ist.org/

[9] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment, in the Proc. of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes In Computer Science; Vol. 2862, Washington, U.S.A, June 2003.

[10] M, Arora, S.K. Das, R. Biswas, A Decentralized Scheduling and Load Balancing Algorithm for Heterogeneous Grid Environments, in Proc. of International Conference on Parallel Processing Workshops (ICPPW'02), Vancouver, British Columbia Canada, August 2002,

[11] Paulsson, M. H¨ubner, J. Becker J.-M. Philippe, C. Gamrat "On- Line Routing of Reconfigurable Functions For Future Self-Adaptive Systems- Investigations within the AETHER project" 17th Iinternatioanl Conference in Field Programable Logic and Applications , FPL07, Amsterdam , Netherlands August 2007.