Can HTCondor manage a condominium cluster? Understanding resource binding parameters

Todd Tannenbaum

May 10, 2013

Abstract

The goal of this project was to conduct a study to understand the basic issues and trade-offs involved in how well the HTCondor system manages scheduling in condominium cluster environment. In particular, the trade-offs for adjusting properties such as the time quanta a resource is bound to a specific user is investigated.

1 Study Background, Justification, and Goals

HTCondor [1] was originally designed to operate across pools of highly heterogeneous non-dedicated desktop machines [2], and to highly leverage preemptive resume techniques in order to meet scheduling goals [3]. Today, HTCondor is increasingly being asked by administrators to manage so-called *condominium clusters*. In a condominium cluster, different user groups purchase the individual servers that make up a homogeneous shared resource dedicated to batch computing. Given two or more user groups, a typical scheduling policy would be to assign $x\%$ to User A and $y\%$ to User B. In HTCondor, a process called *matchmaking* moves resources from one group to another; this study investigates how quickly and effectively this can happen by monitoring both policy enforcement and resource utilization metrics.

In HTCondor matchmaking [4], jobs are submitted into a user agent called a *schedd*. Periodically, a centralized service termed the *Matchmaker* (or alternatively referred to as the *negotiator*) will assign a CPU slot on a machine to a specific user according to user prioritization policies configured by the administrator, at which point the schedd will *claim* that CPU slot on behalf of the user. This claiming step effectively binds that CPU slot to a specific user's workload - that slot will only run jobs submitted from the user that claimed it. In the event that job preemption is disabled, this binding will endure for a minimum of *ClaimWorklife* seconds; the ClaimWorklife is provided by the administrator. Once the amount of time specified by ClaimWorklife has passed, the claim binding the machine to a specific user will be broken at the next job boundary. At this point, the machine is said to be unclaimed, and it will sit idle until it is reassigned by the Matchmaker to another (or potentially back to the same) user.

Parameter	**Variable(s)**	**Meaning**
TimeOfRun	T	Time of the duration of the experiment
ClaimWorklife	C_{WL}	Minimum time a CPU slot must be bound to a specific user; directly corresponds to HTCondor parameter `CLAIM_WORKLIFE`
NegotiatorInterval	N	Interval at which the Matchmaker will bind idle unclaimed CPU slots to a user; directly corresponds to HTCondor parameter `NEGOTIATOR_INTERVAL`
MaxJobRunTime	J_{max}	Maximum job runtime in seconds. Actual job runtime is a uniform distribution between 0 and $J_{max} - 1$ inclusive.
PercentToUserA	$\%A(t)$	Percent of resources desired to go to User A at time t

Table 1: Control Variables

2 Approach and Experimental Methodology

Our approach is a measurement evaluation. Each experimental run starts with two users, User A and User B, and a simple policy: User A should get 100% of the machines. After some time, we change the policy so that User A should get 50% of the machines. Then towards the end of the experiment we put User A back at 100%. We are going to evaluate the system by how closely it can conform to this policy throughout the run.

In the remainder of this section we i) give a detailed explanation of our procedure for running an experimental trial, ii) present the control variables used and measurements taken, and then ii) explain how and which sets of experiments were performed.

2.1 Single Experimental Trial Procedure and Control Variables

For each experimental trial, a personal HTCondor pool is instantiated with one startd configured with 20 CPU slots and one schedd. The Matchmaker is initially configured with a policy to give $x\%$ of the CPU slots to User A and $100 - x\%$ to User B, and an infinitely long workload of jobs is submitted on behalf of both User A and User B. During the run, the value of x may be changed to represent a change in the administrator allocation policy. Throughout the run, metrics are recorded to measure HTCondor's ability to enforce the policy and to use the resources efficiently.

More specifically, given a set of control variable values (see Table 1), each experimental trial followed this procedure:

1. Launch a personal HTCondor instance. Relevant HTCondor configuration settings used for the experiments performed for this study appear in Figure 10 on page 12.
2. Set initial Matchmaker allocation policy for $\%A(0)$ via `condor_userprio` commands for User A and User B. [1]

[1] If you do not know how to do this, see the "How To Configure Fair Share" recipe at `http://bit.ly/19egwhx`

3. With HTCondor's `CLAIM_WORKLIFE` at 0, submit enough jobs charged to User A and User B such that neither user will complete the workload within time T. An example submit description file for $J_{max} = 120$ seconds is shown in Figure 9 on page 11.

4. Let jobs run for J_{max} seconds at a Claim Worklife of 0 before we change it in the next step. This is an attempt prevent our initial experimental start condition from overly influencing monitored dependent variables; in other words, we are trying to avoid an initial condition where all CPU slots are claimed at the same time (in practice, it did not help very much).

5. Reconfig `CLAIM_WORKLIFE` to be C_{WL} via `condor_config_val` and/or `condor_reconfig`, and begin to record the metrics detailed in Section 2.2 by polling the system with `condor_status` every 2 seconds.

6. Adjust the Matchmaker allocation policy via `condor_userprio` at time t whenever $\%A(t) \neq \%A(t-1)$.

7. At time T, stop recording system metrics and shutdown the personal HTCondor instance.

2.2 Measurements and metrics

While each experiment ran, the following measurements were recorded for each CPU slot by regularly polling the system with `condor_status` every ≈2 seconds:

1. The state of the slot (value of startd ClassAd attribute `State`)

2. The time the slot entered the current state (attribute `EnteredCurrentState`)

3. If the slot is in claimed state, the user who owns the claim (attribute `AccountingGroup`)

Given these measurements, the following metrics were derived with the intent they could be graphed and studied:

$\%C(t)$ The percent of CPU slots claimed at time t

$\%C_A(t)$ The percent of CPU slots claimed by User A at time t

$C_{AA}(t)$ The number of CPU slots that transitioned from unclaimed to claimed by User A at time t, and that were also owned by User A the last time they were in claimed state.

$C_{AB}(t)$ The number of CPU slots that transitioned from unclaimed to claimed by User B at time t, and that were also owned by User A the last time they were in claimed state.

$C_{BB}(t)$ The number of CPU slots that transitioned from unclaimed to claimed by User B at time t, and that were also owned by User B the last time they were in claimed state.

$C_{BA}(t)$ The number of CPU slots that transitioned from unclaimed to claimed by User A at time t , and that were also owned by User B the last time they were in claimed state.

$m(t)$ **and** M The number of matches performed by the Matchmaker at time t and over time interval T respectively, defined as

$$m(t) = C_{AA}(t) + C_{AB}(t) + C_{BB}(t) + C_{BA}(t)$$

$$M = \sum_{0}^{T} m(t)$$

$m_{waste}(t)$ **and** M_{waste} When a claim to a slot is broken and subsequently rematched back to the same user, we call this a “wasted” match, since after all the overhead of consulting the Matchmaker resulted in no change to the allocation. Defined as

$$m_{waste}(t) = C_{AA}(t) + C_{BB}(t)$$

$$M_{waste} = \sum_{0}^{T} m_{waste}(t)$$

We also identified two additional useful metrics: *Diff* (d,D) and *Percent Utilization* ($\%U$). Let $A_A(t)$ be the percentage of slots that should be allocated to User A at time t according to administrator policy. We then define

$$d(t) = |\%A_A(t) - \%C_A(t)|$$

$$D = \sum_{0}^{T} d(t)$$

to quantify the ability of the system to enforce the administrators allocation policy. If $D = 0$ the system is able to react instantaneously and perfectly accurately to an allocation policy. The higher the value of D, the more off-target the system is performing -vs- the allocation. Next we define percent utilization to be

$$\%U = \frac{\sum_{0}^{T} \%C(t)}{T}$$

to capture the ability of the system to efficiently utilize the resources.

2.3 Experiments Performed

In the course of the study, ten experiments each consisting of hundreds of trials were run. For each experiment, a small subset of variables (usually just one) listed in Table 1 on page 2 were selected to serve as independent variables while the rest remained as control variables. Next a batch of trials were run, altering the independent variable in each individual trial. Since each trial typically took more than 20 minutes to run, and an experiment usually required hundreds of trials in order to sweep a parameter space for an independent variable, I scripted the above procedure to run an individual trial and submitted batches of experiments into the UW-Madison CS HTCondor pool. I found it peculiarly satisfying to use HTCondor itself to enable research related to HTCondor.

Except when stated otherwise below, the default control variable values used for all ten experiments performed were $T = 1080$ seconds, $C_{WL} = 120$ seconds, $N = 10$ seconds, $J_{max} = 120$ seconds, and

$$\%A(t) = \begin{cases} 100 & : \quad 0 \leq t < \frac{1}{3}T \\ 50 & : \quad \frac{1}{3}T \leq t < \frac{2}{3}T \\ 100 & : \quad \frac{2}{3}T \leq t < T \end{cases}$$

Experiment Set 1 Explore impact of *ClaimWorkLife*.

Experiment 1.0 Run 500 trials where $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment 1.1 Run 500 trials where $T = 4320$, $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment 1.2 Run 500 trials where $N = 5$ seconds, $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment 1.3 Run 500 trials where $N = 20$ seconds, $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment 1.4 Run 500 trials where $N = 40$ seconds, $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment 1.5 Run 500 trials where $J_{max} = 30$ seconds, $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment 1.6 Run 500 trials where $J_{max} = 240$ seconds, $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment 1.7 Run 500 trials where $J_{max} = 480$ seconds, $C_{WL} = \{0, 2, 4, \ldots, 998\}$.

Experiment Set 2 Explore impact of *NegotiatorInterval*.

Experiment 2.0 Run 60 trials, where $N = \{5, 7, 9, \ldots, 123\}$.

Experiment 3 Explore impact of *MaxJobRunTime*.

Experiment 3.0 Run 500 trials, where $J_{max} = \{0, 2, 4, \ldots, 998\}$.

3 Experiment Results

Because lower values for *ClaimWorklife* give the system more opportunities to correct the current allocation by breaking the claim more often, it is a key factor in the ability of HTCondor to react to changes in the allocation policy. In general, lower *ClaimWorklife* values enable the system to more accurately enforce the allocation policy. This is demonstrated in Figure 1 on the next page which shows how *Diff* (D), our metric for allocation policy adherence, changes for different values of *ClaimWorklife*. However, we did discover that setting this value too low had the opposite impact — note how D increases in Figure 1 when C_{WL} drops below 144 and approaches 0. We found that at these low values for C_{WL}, the system was having trouble adhering to the allocation because every time a claim is broken, the slot must sit idle waiting to be re-matched by the Matchmaker. The high rate of claim breaking ultimately resulted in lower overall system utilization; this effect is shown in Figure 3 on page 8. The large drops in D in Figure 1 that begin at time 330 and 690 seconds, or $J_{max}/2$ seconds before each time the allocation policy changed, is primarily due to an initial conditions artifact from our experimental method – specifically the fact that all claims were created within J_{max} seconds of each other (see step 4 in Section 2.1). Figure 2 graphs the same metrics, but *TimeOfRun*

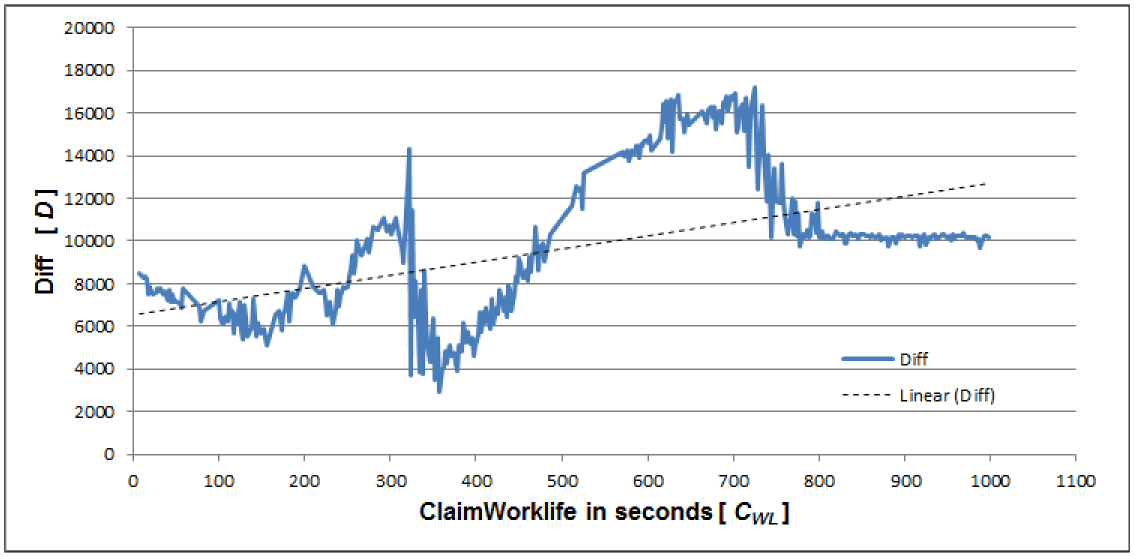

Figure 1: *ClaimWorklife* impact on *Diff* from Experiment 1.0, with linear regression trend-line. $T = 1080$ seconds.

(T) has been increased four-fold; increasing T did not change the fundamental behaviors we had already observed with smaller values for T.

Utilization decreases as the number of unclaimed slots waiting to be matched increases. The time it takes to match an unclaimed slots is the sum of the time the slot must wait for 1) a Matchmaking cycle to begin, plus 2) the time it takes for the Matchmaking cycle to complete, plus 3) the claiming protocol overhead. The time it takes for a Matchmaking cycle to complete is primarily a function of the number of unclaimed slots to match and the number of auto-clusters in the job queue. In our study, because all jobs and machines match each other (i.e. Requirements=True), and because there are only 20 CPU slots in the test pool, the Matchmaking cycle completes in less than a second once started. Thus in our study, the primary influence on how long a slot remains unclaimed is how long it must wait for a Matchmaking cycle to begin, which $\approx N/2$. Therefore we would expect the utilization to decrease as N increases as slots will spend an increased amount of time unclaimed. In addition, we would expect D to increase, since unclaimed slots do not adhere to the desired allocation policy. Both expectations were confirmed by the results from Experiment 2.0 shown in Figure 4 on page 9. The influence of different values of N on a sweep of C_{WL} shown in Figure 5 on page 9, which plots the results from experiment 1.2, 1.3, and 1.4, show that a faster Matchmaking cycle translates to better scheduling policy enforcement.

Because shorter jobs enable more opportunities for the Matchmaker to correct allocation imbalances across users, we observe in Figure 6 on page 10 that shorter job run times result in better policy enforcement (i.e. a lower D value). The zigzag nature of the plot line is because the actual run time of our workload is using a uniform distribution between 0 and J_{max}.

Lastly, we made a couple discoveries by looking at plots of individual trials. In Figure 7 on page 10, the blue (top) line tracks $\%C_A(t)$, the percent of jobs assigned to User A at time t, while the red (bottom) line

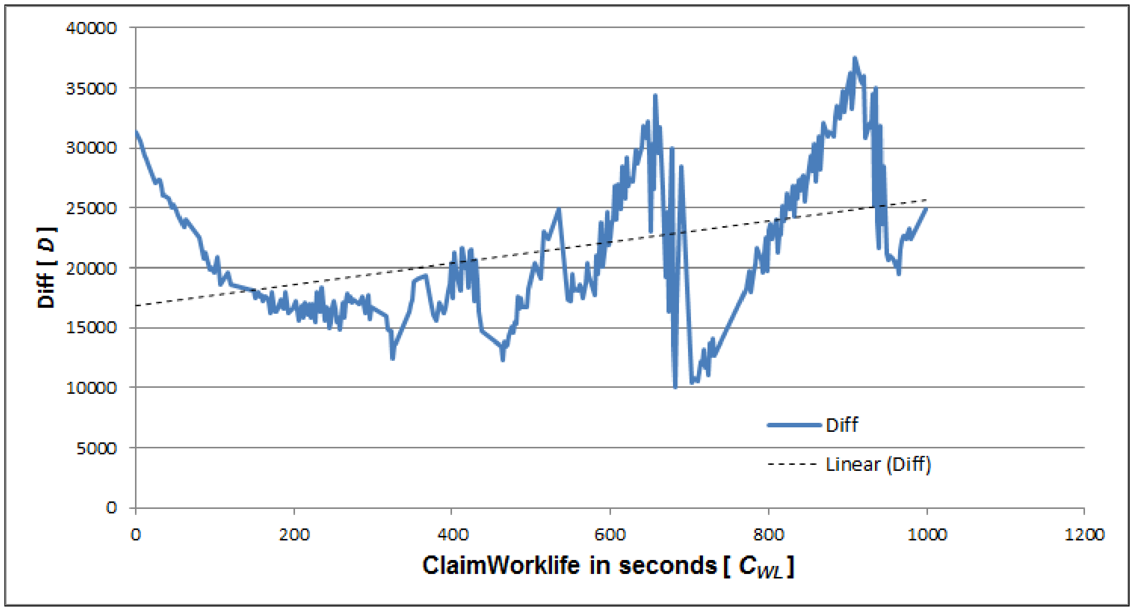

Figure 2: *ClaimWorklife* impact on *Diff* plot from Experiment 1.1, with linear regression trend-line. $T = 4320$ seconds, i.e. four times longer than in Figure 1.

tracks $d(t)$, the *Diff* at time t. Note that the number of jobs drops from $\approx 100\%$ to 50% at a faster rate than it is able to climb back up from 50% to 100%. This behavior is not a anomaly; we found

$$\sum_{T/3}^{2T/3} d(t) < \sum_{2T/3}^{T} d(t)$$

to be true in all experiments, typically by a substantial margin. This behavior can be explained because when the allocation of User A is initially lowered, every claim that is broken can be re-assigned to User B, but when the allocation is then subsequently raised, only a subset of broken claims (those belonging to User B) can be re-assigned.

Figure 8 on page 11 investigates the matchmaking situation for the very same trial, and shows that 91 of 119 matches made during run were wasting time in that the slot was simply given back to the same user. In fact, during this trial, we see that 37 claims were unnecessarily broken before $t = 360$ when the allocation was first changed down from 100%. The price paid for essentially constantly "being ready" to change the current allocation in the event the policy changes in the future is lower system utilization.

When looking at some other trial plots, we noticed that the Matchmaker would occasionally give out one match to User B during a time when User A had a 100% allocation. This is apparently due to a previously undetected round-off error in the Matchmaker's implementation.

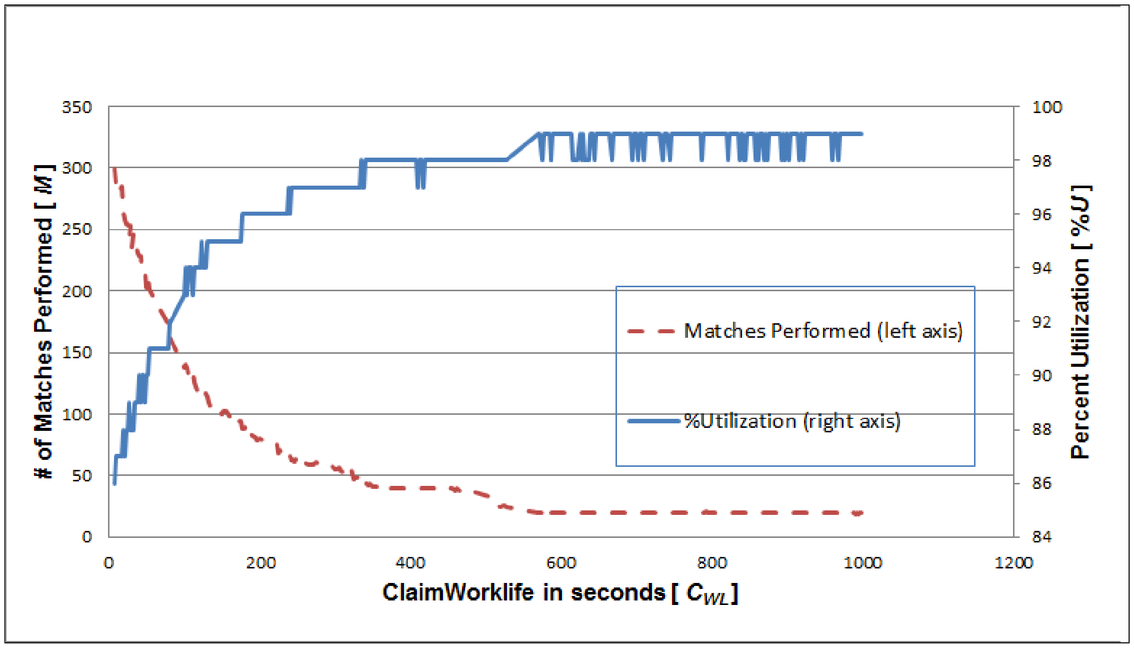

Figure 3: *ClaimWorklife* impact on M and $\%U$ plot from Experiment 1.0

4 Conclusions and Future Work

In conclusion, we found in a condominium cluster situation there is a tension between HTCondor's resource utilization and ability to accurately adhere to a scheduling policy allocation. The administrator can balance utilization vs policy adhesion via setting of the *ClaimWorklife* parameter especially in the event the job workload includes a significant number of jobs that run for a short period of time. What defines "short" here is relative to the time interval between completed negotiation cycles. A five minute job runtime is not considered here to be short if this interval is 20 seconds, but if the negotiator interval is 20 minutes the situation is very different. Given the negative impact on utilization occurring from increased negotiation times, future work could include finding a way to decrease negotiation time in HTCondor especially in situations where the average job runtime is low. One potential solution would be to enable the schedd in certain situations to re-allocate slots between users on its own without waiting for the Matchmaker.

Additional future work could include leveraging the framework developed for this study into HTCondor's regression testing (to catch bugs like the Matchmaker round-off error we discovered), and/or using the metrics identified in Section 2.2 as the basis for a scheduling quality benchmark. Another area of study could investigate a mechanism to enable HTCondor to dynamically adjust *ClaimWorklife* based upon the the current allocation policy in order to reduce the amount of needless matchmaking depicted in Figure 8. Finally, the parameters of this study could be broadened to look at different workloads, pool sizes, and policies.

On a personal note, in the course of performing this study I learned first-hand that even an apparently simplistic system can have a richness of complex interactions that can take significant effort to uncover and understand.

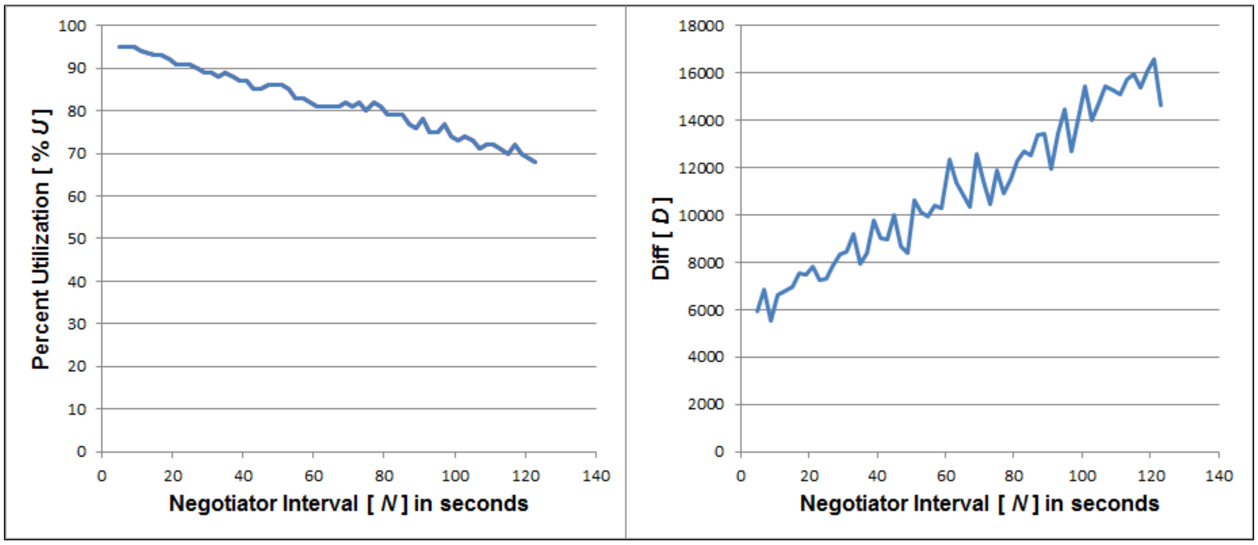

Figure 4: *NegotiatorInterval* (N) impact on $\%U$ (left graph) and D (right graph) plot from Experiment 2.0. As N is typically much smaller than J_{max} in production settings, we chose to examine values for N in the range 5 to $J_{max} + 5 = 125$ seconds.

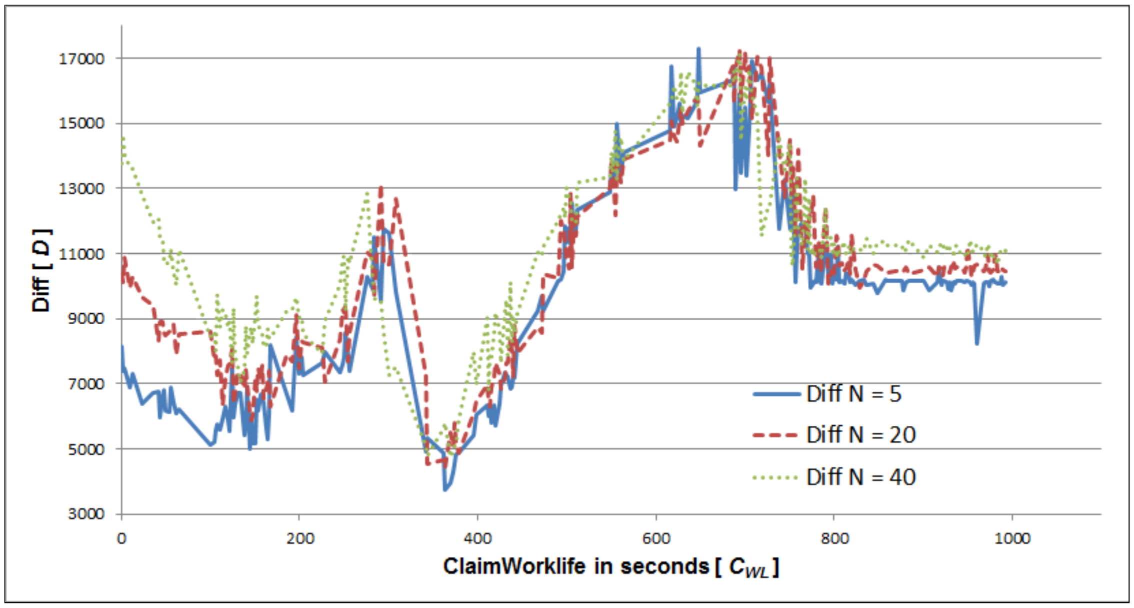

Figure 5: Plots from Experiments 1.2, 1.3, and 1.4 showing the influence of different values for *Negotiator-Interval* (N) upon *Diff* (D) given a range of different values for *ClaimWorklife* (C_{WL}).

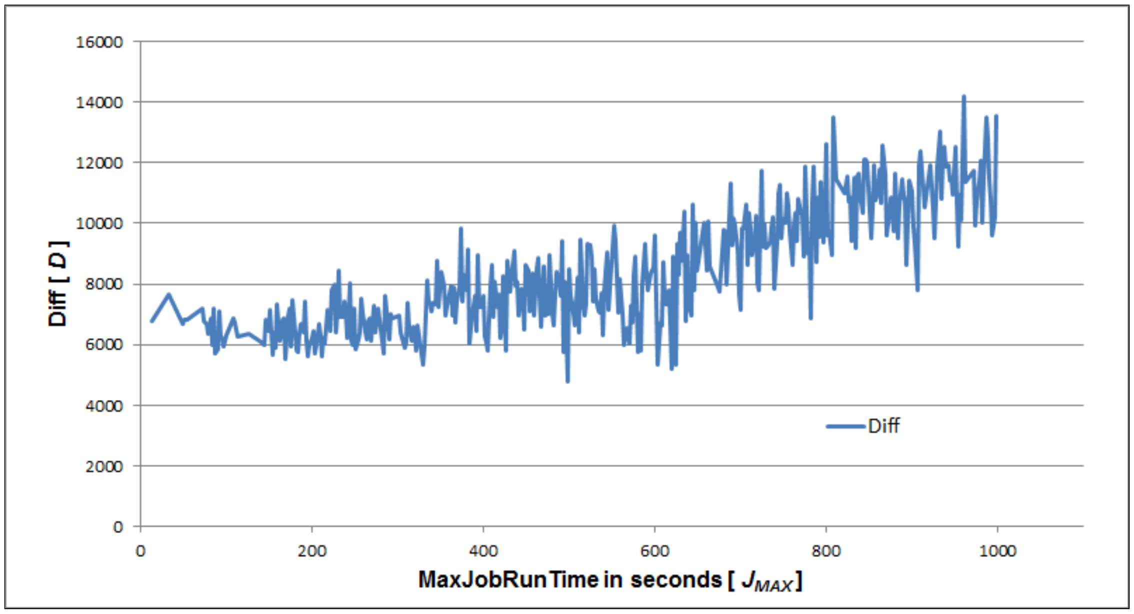

Figure 6: *MaxJobRunTime* impact on *Diff* plot from Experiment 3.0.

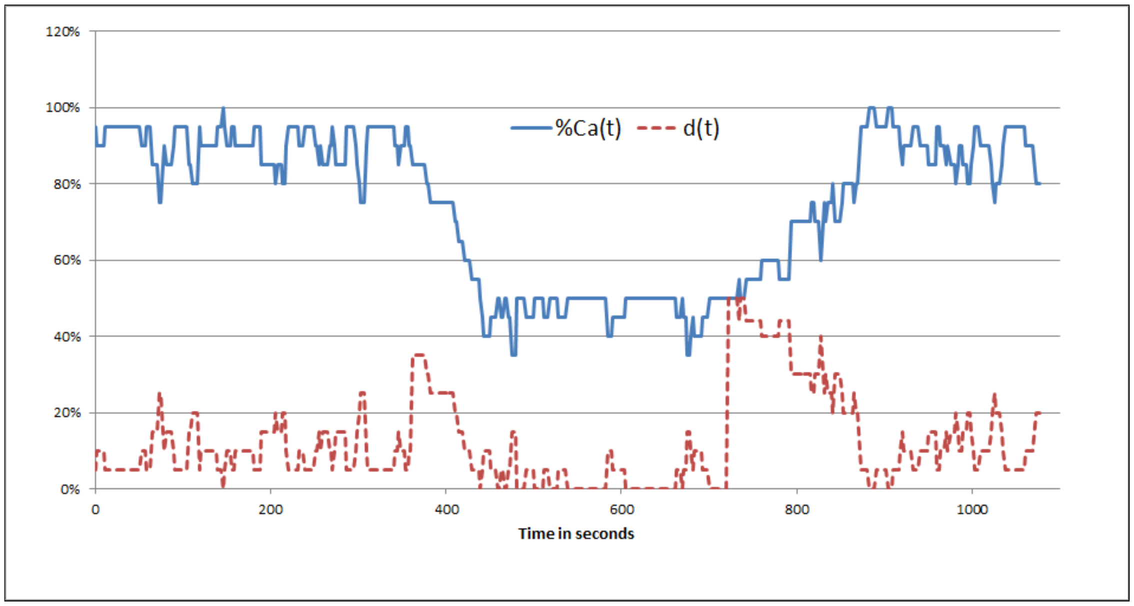

Figure 7: One trial from Experiment 1 ($C_{WL} = 120$) showing $\%C_A(t)$ and $d(t)$

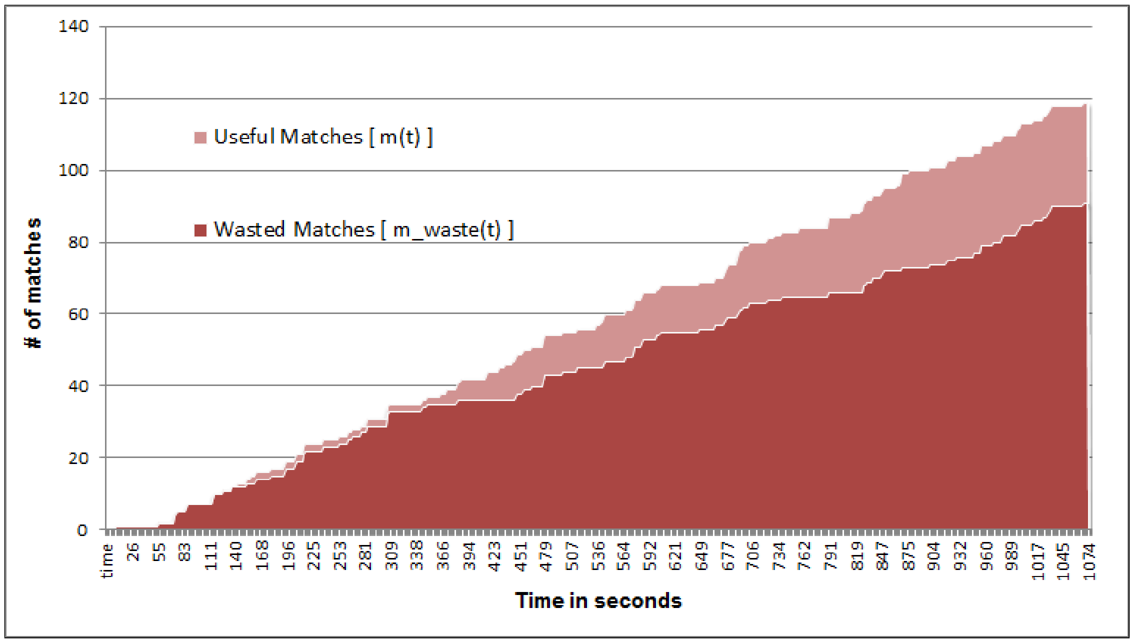

Figure 8: Data from same trial depicted in Figure 7 on the preceding page (Experiment 1 $C_{WL} = 120$) showing contribution of wasted matches ($\sum_0^t m_{waste}(t)$) towards sum of all matches ($\sum_0^t m(t)$)

```
transfer_executable = false
should_transfer_files = NO
universe = vanilla
executable = /bin/sleep
arguments = $RANDOM_INTEGER(1,120)
+AccountingGroup = "userA"
queue 10000
+AccountingGroup = "userB"
queue 10000
```

Figure 9: Sample HTCondor submit description file for $J_{max} = 120$ seconds

```
DAEMON_LIST = MASTER,SCHEDD,COLLECTOR,NEGOTIATOR,STARTD
NUM_CPUS = 20
CLAIM_WORKLIFE = 0
UPDATE_INTERVAL = 2
NEGOTIATOR_INTERVAL = $ENV(NegotiatorInterval)
# Need to also set NEGOTIATOR_CYCLE_DELAY if the
# NEGOTIATOR_INTERVAL is less than 20 seconds
NEGOTIATOR_CYCLE_DELAY = $ENV(NegotiatorInterval)
# Disable automatic adjustment of user priorities
# I.e. turn off the up-down algorithm so only priority factor is used
PRIORITY_HALFLIFE = 1.0e100
# Always match jobs, and never suspend, preempt, or kill a running job
START = TRUE
SUSPEND = FALSE
KILL = FALSE
PREEMPT = FALSE
PREEMPTION_REQUIREMENTS = FALSE
RANK = 0
NEGOTIATOR_CONSIDER_PREEMPTION = FALSE
NEGOTIATOR_INFORM_STARTD = FALSE
# Enable setting of config parameters via condor_config_val command
ALLOW_CONFIG = */*
ENABLE_RUNTIME_CONFIG = TRUE
SETTABLE_ATTRS_CONFIG = *
# Use an ephemeral port for collector instead of the well-known port
COLLECTOR_HOST  = $(CONDOR_HOST):0
# Keep log, spool, execute directories in scratch dir if running
# experiments under HTCondor itself
LOCAL_DIR = $ENV(_CONDOR_SCRATCH_DIR)
```

Figure 10: Relevant personal HTCondor configuration settings used when running trials

References

[1] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[2] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[3] Alain Roy and Miron Livny. Condor and preemptive resume scheduling. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, pages 135–144. Kluwer Academic Publisher, 2003.

[4] Miron Livny and Rajesh Raman. High-throughput resource management. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.